Enhancing Software Quality Assessment: Classifier-Based Reduction of Mutation Test Generation

Zeinab Asghari [1], Bahman Arasteh [2], Abbas Koochari [3]
[1] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.
[2] Department of Computer Engineering, Tabriz Branch, Islamic Azad University, Tabriz, Iran
[3] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.
Email: asghari1677@gmail.com , bahman.arasteh@istinye.edu.tr (Corresponding author) ,koochari@gmail.com

Receive Date: 07 January 2024 Revise Date: 05 March 2024 Accept Date: 13 March 2024

Abstract

Mutation testing could be a capable procedure to assess the quality of test suites. The method of creating mutation testing includes creating a huge number of test cases, which can be computationally costly and time-consuming. This consider proposes a classifier-based approach to diminish the number of created mutation tests that includes preparing a classifier on a set of instruction highlights to decide which ones are error-prone. The classifier is prepared on a dataset of instruction characteristics for identifying the foremost compelling informational for infusing mutants Mutation score is calculated to decide the foremost viable enlightening. The ponder assesses the adequacy of the approach through tests on a few open-source ventures. The comes about appear that the approach is able to decrease the number of produced mutants whereas keeping up high mutation score. This approach has the potential to essentially diminish the computational burden of change testing and make strides the effectiveness of program testing.

Keywords: error propagation, instruction classification, machine learning, software mutation testing

1. Introduction

Software testing is considered an impartible part of the software development process. If the software that is delivered to the customer has an acceptable level of quality, appropriate tests are needed. Software testing will be successful when it can find many errors in the program [1]. To improve the software quality, it is necessary to pay special attention to the software testing step. The effectiveness of a software testing is determined based on the ability of the test case to find faults in a program. If the effectiveness of test cases is higher, the software will be of higher quality [2]. It is noteworthy that the cost of software testing is about 50% of the total cost of the software development process. Therefore, cost and time consumption is two main challenges of this research.

The method to evaluate the effective quality of a test suite is the mutation testing. The underlying idea of the mutation testing is to inject bugs, namely mutants, into the source code of the program. The program containing the injected bugs is called mutant which actually encompass faulty versions of the original program. These syntax changes are usually minor and are designed to reflect common faults that may be present in the original program. A mutant is said to be killed if a test case is found that discriminates between the mutant and the original program. Test suites that kill a large number of mutations are of higher quality than those that kill a small number[3].

These mutants are generated using a mutation tool, which implements mutation operators; rules for how a mutant should be generated from an input program. Table 1 contains only one example of a mutation operator; there are many others.

Mutation testing is empirically more robust than testing metrics such as control-flow-based testing and data-flow-based testing[4]. Despite its effectiveness, several factors make mutation testing expensive and difficult to use experimentally: large sets of mutants that must be run, sometimes many times; creating test cases to kill mutants; number of required tests and equivalent mutants are examples of these factors[5], [6].

Table 1. A Example of Mutating Operation

Program p	Mutant p’

if (a > 0 && b > 0)
return 1;
	if (a > 0 || b > 0)
	return 1;

For a given program p, m signifies a mutant of program p. Review that m is an equivalent mutant on the off chance that m is grammatically diverse from p, but has the same conduct with p. Table 1 appears an case of equivalent mutant produced by changing the operator < of the initial program into the operator ! =. In the event that the statements inside the circle don't alter the value of i, program p and mutant m will deliver indistinguishable yield.

Table 2. A Example of Equivalent Mutation

Program P	Equivalent Mutant m
for (int i = 0; i !< 10; i + +) { (the value of i is not changed)}	for (int i = 0; i ! = 10; i + +) { (the value of i is not changed)}

Mutations operators adjust the program beneath test to form mutants. For case, an arithmetic operator would alter the expression (a + b) to (a ∗b), (a −b), and (a / b). Mutation operators utilize fault scientific classifications that are ordinarily based on ponders of issues in genuine programs. Change operators are applied to a program P to form a set of mutants M . Each test t in a test set T is run against each mutant m,, m ∈ M . If m (t) _ = P (t) for some t , then we say that t has killed m . If not, the tester should find a test that kills m . If m and P are equivalent, then P (t) = m (t) for all possible test cases.[7] .

An equivalent mutant plays the role of a parasite in the testing process. Indeed, while it is expected to be killable, it remains always live even worse, a tedious effort could be uselessly dedicated to improving tests with no hope of killing it Consequently, mutation testing should be able to detect and exclude these mutations. However, the issue of functional equivalence of programs is undecidable [*8*]. If the analysis is done manually, it will be very tedious. It has been found empirically that the identification of an equivalent mutation in a real-world application takes approximately 15 minutes. [9]. Because there are many equivalent mutations in real applications, the cost of mutation application will be high.

In this paper, all program instructions are analyzed and a set of program code level characteristics that contribute to error propagation rate are listed. The instructions with low rate are assigned to the supervised machine-learning algorithm for classification. Finally, based on classification, instructions with low efficiency are removed from the program so that the amount of mutations generated is minimized. Classification is one of the ways to increase the accuracy of classification, which can be effective in improving the classification process.

2. Related Works

Computation costs of mutation testing are one of the challenging research problems in this field of study. Researchers have proposed different techniques for solving this problem. In the following some of the key techniques and methods are discussed:

Authors in [10] investigated the idea of selective mutation, which uses only the most critical mutation operators. in fact this method selects a subset of the mutation operators[11]. Offutt and Zapf [12] extended the selective mutation idea, which allows testers to perform approximate mutation testing. They demonstrated that reducing the number of mutants decreases the testing costs while providing coverage that is almost as strong as non-selective mutation. The selective set of mutation operators (appropriately modified for Java) were implemented for Java in muJava [13]. Kaminski and others[14] further showed that only three mutants out of the seven created by the relational operator replacement operator (ROR) are needed. Mutant sampling is on of the most straightforward strategies which selects a subset of the mutants randomly. Mutant random sampling was one of the first attempts to mutant reduction. Higher order mutants (HOMs) first introduced by Jia & Harman [15], combine the changes from multiple first-order mutants (FOMs), i.e. single statement mutants, into one mutant [16]. It is also possible to generate HOMs that are subsuming; the test cases that kill a subsuming HOM also kill every FOM that it is generated from. Consequently, using HOMs also allows for the execution time of mutation testing and analysis to be reduced, since if a subsuming HOM is killed, each of its constituent FOMs are as well. Antonio and vergilio [9] display comes about of a mapping consider, by synthesizing characteristics of the HOM Testing approaches, HOM generation techniques, assessment viewpoints, patterns and investigate openings. Strong Mutation is regularly alluded as conventional Mutation Testing. That's , it is the detailing initially proposed by DeMillo et al. [17]. In Strong Mutation, for a given program p, a mutant m of program p is said to be murdered as it were on the off chance that mutant m gives a different output from the initial program p. To optimize the execution of the Strong Mutation, Howden [8] proposed Weak Mutation. In Weak Mutation, a program p is expected to be built from a set of components C = {c1, ..., cn}. Assume mutant m is made by changing component cm, mutant m is said to be killed on the off chance that any execution of component cm is diverse from mutant m. Firm Mutation was to begin with proposed by Woodward and Halewood [18]. The thought of Firm Mutation is to overcome the impediments of both weak and strong mutations by giving a continuum of middle conceivable outcomes. The thought of Mutant Clustering was first proposed in Hussain's ace proposal [19]. Rather than selecting mutants randomly, Mutant Clustering chooses a subset of mutants utilizing clustering calculations. The method of Mutation Clustering starts from creating all to begin with arrange mutants. A clustering calculation is at that point connected to classify the primary arrange mutants into diverse clusters based on the killable test cases. Each mutant within the same cluster is ensured to be killed by a comparable set of test cases. Combefis and Schils has benefited unsupervised clustering to aid the assessment of large quantities of solution programs [20].

3. Propsed MethodE

There are a series of instructions of the program code that do not have much effect on the output of the program. But this does not mean that these instructions should be completely removed from the program. The selection of this type of instructions is based on the error propagation rate (ep-rate). The instructions with the lowest level of effectiveness do not necessarily mean their complete removal from the program code. In some cases, keeping that instruction in the program may increase the mutation score in the program. In this paper, we use the programs with java languages. The reason for choosing Java language to check mutations in the proposed method is the existence of limited platforms in the field of mutation production. Mujava, as the most powerful platform in this field, has made the flexibility and applicability of the proposed method limited to Java language only.

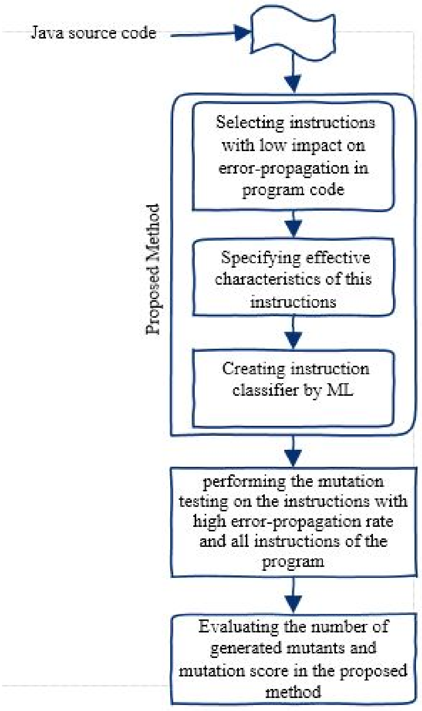

Java source code
Proposed Method
Selecting instructions with low impact on error-propagation in program code
Specifying effective characteristics of this instructions
Creating instruction classifier by ML
performing the mutation testing on the instructions with high error-propagation rate and all instructions of the program
Evaluating the number of generated mutants and mutation score in the proposed method

Fig. 1. Steps of the proposed method

The supervised machine learning algorithms were used to classify instructions. Those machine learning algorithms are Gradient Boosted Trees, Decision Tree, Multi-Layer Perceptron, and Random Forest and Neural Networks. After classifying the instructions mutation operators are applied on selective instructions. The proposed method is depicted in Fig.1.

In any program, there are many instructions, the nature of each instruction is different from the other . This difference is determined based on a series of criteria and characteristics. If we want to check Java language programs, there are instructions in programs that removing those instructions will not affect the output of the program. We call this *Z-instruction*. For example, notification and print instructions. But in the discussion related to the software mutation test, the case is different. In this case, Z-instructions that have low effectiveness in the output of the program have a different effect on the mutation score. Among Z-instructions, sometimes not deleting that instruction will increase the mutation score. This is because some of the characteristics of those instructions are different from the rest of the instructions and the remaining of that instruction can somehow change the number of kill and live mutants. So, the purpose of this part of the method is to separate notification and print instructions and classify Z-instructions. For example, Fig.2 shows the part of java *Prime* program, the total number of Z-instructions is 7. While it can be proven that the effect of these 8 instructions on the mutation score is different. To prove this, we need to use a classification with supervised machine learning algorithms. Classification of instructions can help in dividing these types of instructions based on effecting. To identify instructions with low impact on program output, we need to be able to separate low impact instructions from other instructions based on a series of features. The values associated with each characteristic in the table have been assigned. Each entry takes different values based on the instruction in the program code. This characteristic based on which the level of effectiveness can be determined are as follows:

i. *Average number of executions*: This property specifies the average number of executions of each instruction for test data.

ii. *Number of variables*: Any instruction can declare a variable or use a variable based on its application in the program. In Z- instructions, this feature has a great impact to determine the level of the instruction. The sum of the test variables and other variables is the *number of variables*.

iii. *Test variable*: In order to be able to identify the behavior of the program, we need to use test data. The test data should be edge-coverage. that is, they can cover all edges of the program as much as possible based on the edge coverage property. Running the majority of edges allows us to identify the behavior of individual instruction. The variables in the instructions can be the type of variables used in the test data. If these types of variables are available in the Z instructions, the value of this column in the data table will be equal to 1.

iv. *Other variable:* Any variable other than the variable used by the test data is placed in this class. *Control dependency:* This feature represents the number of next instructions which has control dependency on the result of the current instruction.

v. *Static variable:* Static variable is variable which belongs to the class and initialized only once at the start of the execution. It is a variable which belongs to the class. Static variables are initialized only once, at the start of the execution.

vi. *Nesting level:* The nesting level of an instruction shows the accessibility of the instruction. If the instruction is not in an if instruction, its nesting level is 0; if it is in an if instruction, then its nesting level is 1.

Equation.1 shows the ep-rate. The ep-rate of each instruction in a program has been measured by 100 executions in the presence of the injected mutant. The number of times the program fails divided by 100 indicates the ep-rate rate of an instruction.

$$\text{Errro} - \text{Propagation Rate} = \frac{\text{Number of failure}}{\text{Total Number of Execution}} * 100 \qquad (1)$$

```
public class Prime {
public int prime(int n) {
int i=1;
boolean   isPrime = true;
System.out.print(i+ " is a secondary variable\n");
System.out.println(n+ " must be a positive
number");
if (n == 0 || n == 1) {
isPrime= false;
System.out.printf(n+ " is Not Prime");
return 0;
} else {
for (i = 2; i <= n/2; i+=1) {
if (n % i == 0) {
isPrime = false;
System.out.printf ("'%S' %n"," is Not prime");
break;
.
.
.
```

Fig. .2. A summary view of Prime java program

The MuJava tool is used instructions ep-rates and quantify the *Rank* feature.this feature represents ep-rate of Z-instructions.The category of ep-rate is shown in table 3.

Table 3. Error-propagation rate of Z-instructions

propagation rate	Category
11% - 30%	A
0% - 10%	B

4. Results and Discussion

A series of mutation testing experiments has been performed in order to measure the ep-rate of Z-instructions. We must use machine learning algorithms to create a classification of benchmark programs. For this purpose, it is necessary to use the ep-rate. The supervised machine learning algorithms used in this section are: AutoMLP, Neural Networks, Random Forest and naive bayes. The performance of the created classifier has been compared with each other. Table 4 shows the *Prime* program data required for the classification of machine learning algorithms. These data are extracted for all benchmark programs and will be used as input for ML algorithms. Indeed the last column is the dependent variable and the other features are independent variables that are used in the training stage of the machine learning algorithm. In this paper, we use the RapidMiner tool for implement data classification. Table 6 shows the performance of the created classifier by different ML algorithms in terms of accuracy, precision, recall, and kappa.

The dataset prepared in table 4 was used to train the ML algorithm and the created classifier by the ML algorithms has been tested in the same way (k-fold). The created classifier is a multi-class classifier; the outputs of the classifier are shown in table 3. Every Z-instruction in a 2-class classification must be sorted into one of two categories. Given a set of program Z-instructions at the source code level, the generated classifier must determine which category (A, B) each Z-instruction belongs to. Indeed, the created classifier takes the features of an instruction and predicts its classes in terms of its error-propagation rate. This stage of the proposed method has been implemented in the RapidMiner tool set. RapidMiner includes an extensive data analysis library and it is one of the most frequently used tools for data analysis and data mining applications. Table 5 shows the details of benchmarks programs used in proposed method. These programs have been used abundantly in the experiments of various articles. In the proposed method, it has not been possible to check real-world huge programs due to checking programs at the instruction level. Because the mutations that are created during the review process of the proposed method in medium and small programs show a significant increase. For example, in the calculator program, with 31 lines of code, 112 mutants have been created, which is a relatively large number for checking mutations and classifying them.

Table 4. The values of the features for the Prime benchmark program

Runtime Average	Number of variables	Test variable	Other variable	Control dependency	Static variable	Nesting Level	Rank
1	1	1	0	0	1	0	A
1	1	1	0	1	1	0	B
1	1	1	0	0	1	1	B
3	1	1	0	1	1	2	A
2	1	0	1	0	0	1	A
1	1	0	1	0	1	0	A
1	1	1	0	0	0	0	B

Table 6 shows the performance of the created classifier by different ML algorithms in terms of accuracy, precision, recall, and kappa. There is potential limitations or drawbacks of using a classifier-based approach such this approach. The risk of misclassifying in supervised machine learning algorithms occurs when the algorithm incorrectly assigns a label or category to a data point. This can happen due to various reasons such as insufficient or biased training data, incorrect feature selection, or inappropriate model selection. Misclassification can lead to inaccurate results. Over-reliance on supervised machine learning algorithms without proper human oversight can also lead to potential drawbacks.

For example, if the algorithm is not monitored and updated regularly, it may become outdated and less accurate over time. Additionally, the algorithm may not be able to handle new or unexpected data that was not included in the training data. This can lead to incorrect predictions or decisions, which can have negative consequences in real-world applications.

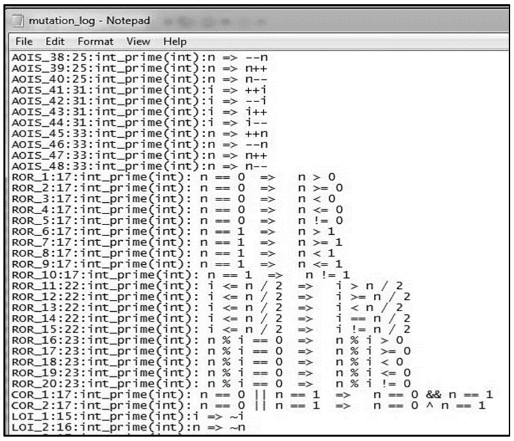
Table 5. Benchmark programs

program	LOC	no. of Z_instruction	program description
Prime	34	7	Determines if it is a prime number
Perfect	21	5	Determines if it is a perfect number
Factorial	26	5	Determining factorial number
Triangle	28	5	Determining triangle type
Calculator	31	5	Building a calculator

Table 6. The output of different ML algorithms

Name of ML algorithm	Accuracy	Kappa	Recall	Precision
MLP	100%	1	100%	100%
Naive bayes	100%	1	100%	100%
Random Forest	100%	1	100%	100%
NN	96.67%	0.927	95.45%	97.50%

After the classification is done by machine learning algorithms, it is time to create mutations. Now, based on the features of each leveled instruction, test mutations are created. For example, the mutations created for the Prime program are shown in Figure 3. After generating mutant programs , Mujava uses the Junit tool for executing the test and evaluating the mutation score. in this level , the error propagation rate for each instruction is evaluated by the proposed method. Then, the instructions with a high error propagation rate are detected. Our first priority the Z-instructions with a high error-propagation rate were subjected. The Z-instructions with a low rate of error-propagation are subjected to mutation operators in the next priority. The status of the created mutants is then looked into in terms of being alive or killed. After that, it is calculated how many mutations were made on the Z-instructions with the highest mistake propagation rate and the average mutation score. Finally, the obtained results are compared and contrasted with those of the previous related works. Because in this article, the programs available in this method have been checked at the instruction level and each instruction has been classified in a very clear and precise way, also the classification of the program instructions has been done based on machine learning methods, so this article is somehow It is unique and it can even be said that it is considered superior to other existing methods.



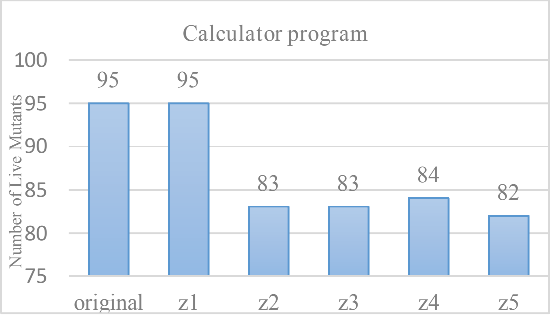
```
AOIS_38:25:int_prime(int):n => --n
AOIS_39:25:int_prime(int):n => n++
AOIS_40:25:int_prime(int):n => n--
AOIS_41:31:int_prime(int):i => ++i
AOIS_42:31:int_prime(int):i => --i
AOIS_43:31:int_prime(int):i => i++
AOIS_44:31:int_prime(int):i => i--
AOIS_45:33:int_prime(int):n => ++n
AOIS_46:33:int_prime(int):n => --n
AOIS_47:33:int_prime(int):n => n++
AOIS_48:33:int_prime(int):n => n--
ROR_1:17:int_prime(int): n == 0  =>   n > 0
ROR_2:17:int_prime(int): n == 0  =>   n >= 0
ROR_3:17:int_prime(int): n == 0  =>   n < 0
ROR_4:17:int_prime(int): n == 0  =>   n <= 0
ROR_5:17:int_prime(int): n == 0  =>   n != 0
ROR_6:17:int_prime(int): n == 1  =>   n > 1
ROR_7:17:int_prime(int): n == 1  =>   n >= 1
ROR_8:17:int_prime(int): n == 1  =>   n < 1
ROR_9:17:int_prime(int): n == 1  =>   n <= 1
ROR_10:17:int_prime(int): n == 1  =>   n != 1
ROR_11:22:int_prime(int): i <= n / 2  =>   i > n / 2
ROR_12:22:int_prime(int): i <= n / 2  =>   i >= n / 2
ROR_13:22:int_prime(int): i <= n / 2  =>   i < n / 2
ROR_14:22:int_prime(int): i <= n / 2  =>   i == n / 2
ROR_15:22:int_prime(int): i <= n / 2  =>   i != n / 2
ROR_16:23:int_prime(int): n % i == 0  =>   n % i > 0
ROR_17:23:int_prime(int): n % i == 0  =>   n % i >= 0
ROR_18:23:int_prime(int): n % i == 0  =>   n % i < 0
ROR_19:23:int_prime(int): n % i == 0  =>   n % i <= 0
ROR_20:23:int_prime(int): n % i == 0  =>   n % i != 0
COR_1:17:int_prime(int): n == 0 || n == 1  =>   n == 0 && n == 1
COR_2:17:int_prime(int): n == 0 || n == 1  =>   n == 0 ^ n == 1
LOI_1:15:int_prime(int):i => ~i
LOI_2:16:int_prime(int):n => ~n
```

Fig. 3. A view of operators for prime program

Table 7. Generated mutants for all benchmarks by the MuJava tool

program name	Total mutants	number of mutants after deleting Z-instructions	number of mutants for A-level of Z-instructions	number of mutants for B-level of Z-instructions
calculator	112	58	12	42
Prime	112	47	10	55
Perfect	70	54	23	38
Triangle	213	169	14	29
Factorial	78	53	9	16

In this paper, we cannot compare our proposed method with previous methods. The reason of lack of comparative analysis between different classifier-based approaches is that the previous existing methods which are based on classification did not analyze the programs at the instruction level, so it is not possible to compare the proposed method with the existing methods. In fact, the previous existing methods have injected mutations at the block level or they have not done the classification based on machine learning classification methods. The following graphs show the results of tests performed on 5 benchmark programs using the Mujava tool. In these experiments, the average results of 5 test data series have been used in the programs. As you can see in the figures, on the horizontal axis, *original* is the original program, z1 is the modified version of the original program by removing the first instruction of level D, and similarly z5 is the modified version of the original program by removing the last instruction of level D. In fact, as many instructions as level D, we will have charts on the horizontal level. Figure 4 shows the number of live mutations for the *calculator* program. In this program, there are 5 level D instructions. As shown in the figure, there are instructions that, by removing that instruction, the number of live mutations has decreased significantly. Similarly, Figure 5 shows the average kill mutants per 5 level D instructions. In Figure 6, the results obtained for score mutation are shown.



Number of Live mutants in the mutation test performed on the calculator program

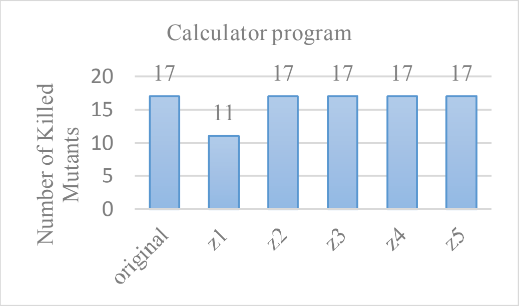


Fig. 4. Number of killed mutants in the mutation test performed on the calculator program

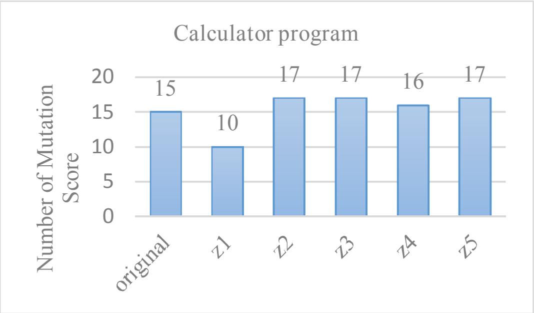


Fig. 5. Number of mutation score in the mutation test performed on the calculator program

Figure 7 shows the results of live mutations on Prime program. According to the figure, this program has 6 instructions in D level. Compared to the original program, by removing the level D instruction in the modified z6 program, the number of live mutations has been significantly reduced. In Figure 8, the amount of kill mutants shows that only by removing the instruction into z1 program, the number of kill mutants has decreased significantly. Figure 9 shows the mutation score for the Prime program. The results indicate that the deletion of the z4 instruction has reduced the mutation score. It can be concluded that if the instruction in z4 program is not removed from the program, the mutation score can be kept at an acceptable level.

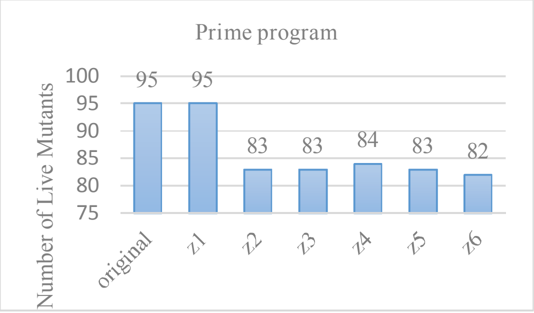


Fig. 6. Number of live mutants in the mutation test performed on the Prime program

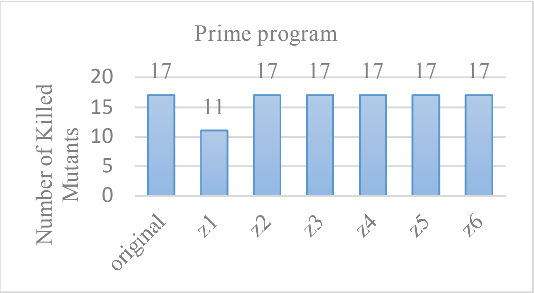


Fig. 7. Number of killed mutants in the mutation test performed on the Prime program

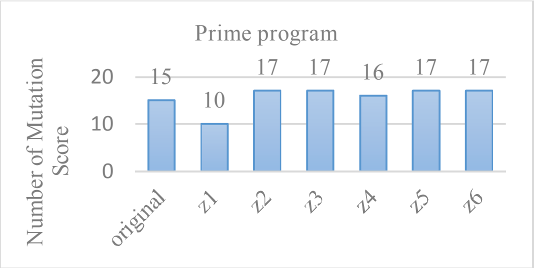


Fig. 8. Number of mutation score in the mutation test performed on the Prime program

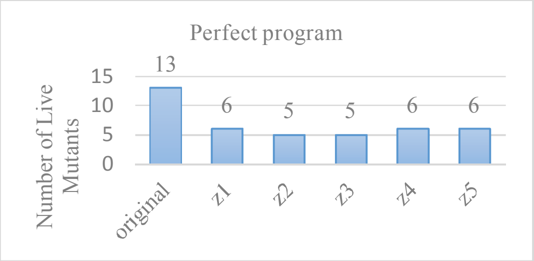


Fig. 9. Number of live mutants in the mutation test performed on the Perfect program

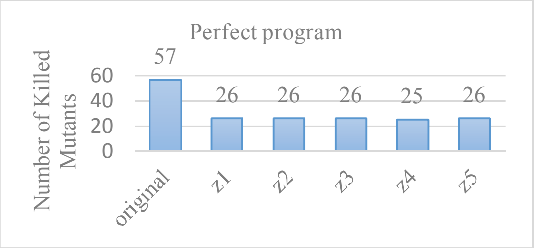


Fig. 10. Number of killed mutants in the mutation test performed on the Perfect program

Figure 10 shows the changes made to the number of live mutants on 5 different versions of the Perfect program. The results show that removing the D level instructions in the two created programs z2 and z3 has reduced the number of live mutants. Figure 11 shows that the z4 program has fewer killed mutants than the rest of the programs. In fact, the level D instruction in this program should not be removed from the main program to achieve optimal results. Figure 12 also shows the bounce score for the Perfect program. As you can see in the figure, the two programs z2 and z3 have a better situation than the rest of the programs compared to the original program. In fact, the D-level instruction sin these two programs should be removed from the program.

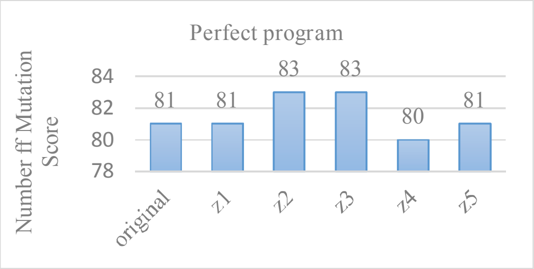


Fig. 11. Number of mutation score in the mutation test performed on the Perfect program

Figure 13 shows the test results on the *Triangle* program. As you can see, the number of live mutants has decreased in the first version of the original program. But in the rest of the versions produced from the original program, the reduction of live mutants is more noticeable. Likewise, in Figure 14, the number of kill mutants in the first version of the main program has been significantly reduced. This shows that it is better not to delete the level D instruction in the z1 program. The reduction of the mutation score in the first version of the original program in Figure 15 also proves this claim.

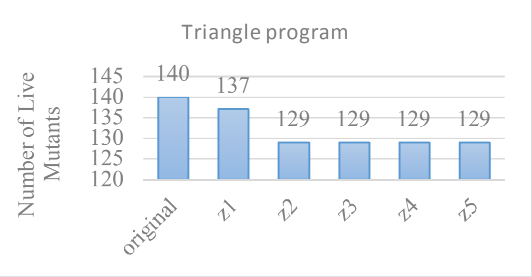


Fig. 12. Number of live mutants in the mutation test performed on the Triangle program

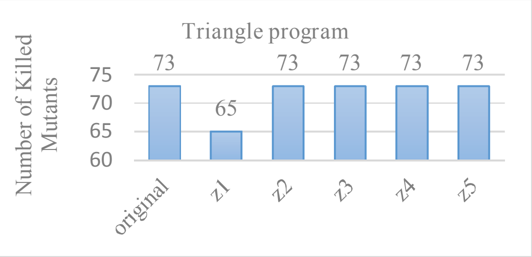


Fig. 13. Number of killed mutants in the mutation test performed on the Triangle program

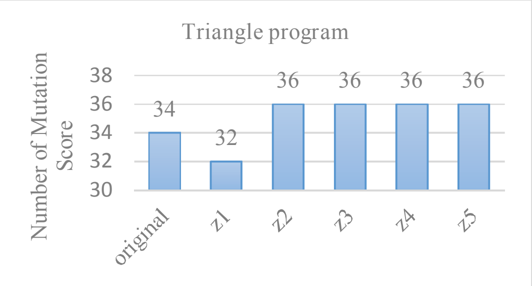


Fig. 14. Number of mutation score in the mutation test performed on the Triangle program

In Figure 16, the two instructions in programs z1, z2 of the Factorial program have been identified as effective instructions in the main program from level D, and it is better to remove them from the program. In Figure 17, except for the instruction in the z3 program, the rest of the instructions have similar conditions, that is, by removing these instructions, the number of killed mutants has increased. And finally, Figure 18 shows an increase in the mutation score in all programs, but in the z3 program, the increase is not significant.

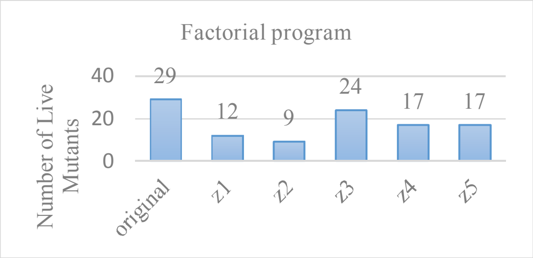


Fig. 15. Number of live mutants in the mutation test performed on the Factorial program

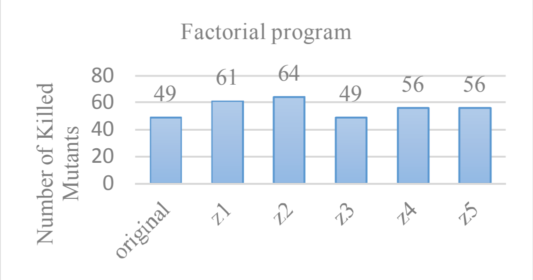


Fig. 16. Number of killed mutants in the mutation test performed on the Factorial program

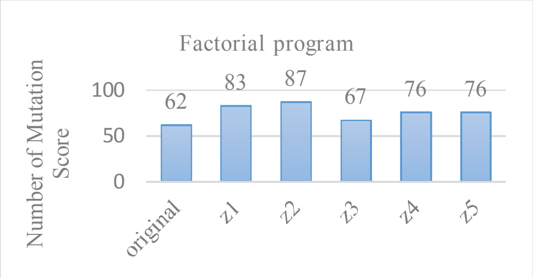


Fig.17. Number of mutation score in the mutation test performed on the Factorial program

This article examines the level D instructions available at the level of the program code and we showed that some instructions at this level may increase the number of generated mutants and decrease the speed of the program by removing the instruction. Therefore, in some cases, it will not be necessary to delete these instructions. In this article, we have shown that the rate of error propagation rate in some instructions of the program is higher than in other instructions, and in some instructions this rate is lower. In instructions that have a lower error propagation rate, two categories of instructions have been identified. Removing the second level from these instructions has increased the mutation score and reduced the number of live mutants. The scalability of the proposed approach is limited to the size and complexity of software systems to which it is applied. Considering that the number of mutations generated in the examined programs has a direct relationship with the number of program lines and the complexity of the program instructions, so this creates a limitation for the author to select only programs to generate mutants in which The number of created mutants should be reasonably small so that the speed of program execution and the quality of instruction execution do not decrease.

Refrences

[1] Nasrin Shomali and Bahman Arasteh “Mutation reduction in software mutation testing using firefly optimization algorithm”; Data Technologies and Applications Emerald Publishing Limited 2514-9288 DOI 10.1108/DTA-08-2019-0140 11 April 2020

[2] Asghari, Zeinab, Bahman Arasteh, and Abbas Koochari. "Effective Software Mutation-Test Using Program Instructions Classification." Journal of Electronic Testing (2024): 1-27,DOI:10.1007/s10836-023-06089-0.

[3] Beller, Moritz, et al. "What it would take to use mutation testing in industry—a study at facebook." 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021,DOI: 10.1109/ICSE-SEIP52600.2021.00036.

[4] P. G. Frankl, S. N. Weiss, and C. Hu, “All-uses vs mutation testing: an experimental comparison of effectiveness” Journal of Systems and Software, vol. 38, no. 3, pp. 235–253, 1997,DOI: https://doi.org/10.1016/S0164-1212(96)00154-9.

[5] Offutt, A.J. , Untch, R.H. , 2000. “Mutation 2000: uniting the orthogonal”.In: Proceed- ings of the Mutation 20 0 0 Symposium. Kluwer Academic Publishers, San Jose, CA , USA , pp. 34–44,DOI:https://doi.org/10.1007/978-1-4757-5939-6_7.

[6] Alessandro Viola Pizzoleto et al.2019.”A systematic literature review of techniques and metrics to reduce the cost of mutation testing”, The Journal of Systems and Software 157 (2019) 110388, DOI: 10.1016/j.jss.2019.07.100.

[7] Mateo, Pedro Reales, and Macario Polo Usaola. "Reducing mutation costs through uncovered mutants." Software Testing, Verification and Reliability 25.5-7 (2015): 464-489, DOI: 10.1002/stvr.1534.

[8] T.A. Budd, D. Angluin, “Two notions of correctness and their relation to testing”, Acta Inform. 18(1) (1982) 31–45, DOI:10.1007/BF00625279.

[9] D. Schuler, A. Zeller,” Covering and uncovering equivalent mutants”, Softw. Test. Verif. Reliab. (2012), DOI: 10.1002/stvr.1473.

[10] W. E. Wong and A. P. Mathur, “Reducing the cost of mutation testing: An empirical study,” Journal of Systems and Software, vol. 31, no. 3,pp. 185–196, 1995, DOI: 10.1016/0164-1212(94)00098-0.

[11] Ellen Francine Barbosa, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. “Toward the determination of sufficient mutant operators for C.” In: Software:Testing Verification and Reliability 11 (2001), DOI: 10.1002/stvr.226.

[12] A.J. Offutt, G. Rothermel, and C. Zapf. “An experimental evaluation of selective mutation”. In 15th International Conference on Software Engineering, pages 100-107,May 1993, DOI: 10.1109/ICSE.1993.346062.

[13] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. “MuJava : An automated class mutation system”. Wiley's Software Testing, Veriffcation, and Reliability, 15(2):97-133,June 2005, DOI: 10.1002/stvr.308.

[14] Gary Kaminski, Paul Ammann, and Jeff Offutt. “Improving logic-based testing”. Journal of Systems and Software, 86(8):2002-2012, August, DOI: 10.1016/j.jss.2012.08.024.

[15] Y. Jia and M. Harman. “Constructing Subtle Faults Using Higher Order Mutation Testing”. In: Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on. 2008, pp. 249–258, DOI: 10.1109/SCAM.2008.36.

[16] Jia, Yue, and Mark Harman. "Higher order mutation testing." Information and Software Technology 51.10 (2009): 1379-1393.

[17] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. “Hints on test data selection: Help for the practicing programmer”. Computer, 11(4):34-41, April 1978, DOI: 10.1109/C-M.1978.218136.

[18] M. R. Woodward and K. Halewood, “From Weak to Strong, Dead or Alive “ an Analysis of Some Mutation testing Issues,” in Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA’88). Banff Albert, Canada: IEEE Computer Society, July 1988,pp. 152–158, DOI: 10.1109/WST.1988.5370.

[19] S. Hussain, “Mutation Clustering,” Masters Thesis, King’s College London, Strand, London, 2008.

[20] Sebastien Combefis and Arnaud Schils.” Automatic programming error class identification with code plagiarism-based clustering”. In Proc. 2ndInt. Code Hunt Work. Educ. Softw. Eng. - CHESE 2016, pages 1-6,New York, New York, USA, 2016. ACM Press, DOI: 10.1145/2993270.2993271.