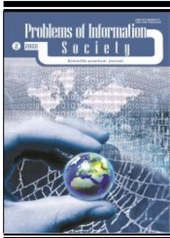

www.jpis.az

13 (2)
2022

Automatic generation of test cases for error detection using the extended Imperialist Competitive Algorithm

***Shahrokh Jalilian*[1]*, Shafagat J. Mahmudova*[2]**

[1]Space Research Institute, Fourteenth Saadat Abad str., 74, Tehran, Iran

[2]Azerbaijan National Academy of Sciences, Institute of Information Technology, 9A, B. Vahabzade str., AZ1141 Baku, Azerbaijan
[1]shjalilian@gmail.com, [2]shafagat7@gmail.com

orcid.org/0000-0002-3161-5425[1]*, orcid.org/0000-0003-1817-0756*[2]

ARTICLE INFO

http://doi.org/10.25045/jpis.v13.i2.06

Article history:
Received 10 January 2022
Received in revised form
14 mart 2021
Accepted 16 May 2022

Keywords:
Automatic Generation
Imperialist Competitive Algorithm
Error Detection
Software Test
Reliability

ABSTRACT

As computing technology progresses, computer systems and their activity domain are becoming widespread, and software projects are becoming complicated in the current society. Software testing is time-consuming and expensive. It aims at validating software functional and non-functional requirements, including software performance. During the test stages, first, it is specified whether software elements perform their tasks accurately and create correct output. While in software testing at the program code level, we can text all circles and lines of program and conditional parts of the program while there needs data in these tests which can test all these cases and cold pass the program lines with the most coating that is one of the most challenging problems in this type of software tests. Therefore, Imperialist Competitive Algorithm, an advanced algorithm, is considered for producing optimal test data for finding errors in programs. Practical results and evaluating the proposed method with other methods indicate the presented algorithm's excellence.

1. Introduction

The main goal of software testing is to estimate and evaluate the quality of the product in each stage of software development. Software evaluation is performed for different reasons, of which we can mention estimating quality, reliability, ease in keeping and supporting, safety, etc. Testing software can be performed on different levels, such as unit testing, integrity, system, and acceptance. One of the primary stages of testing a system is unit testing [1], in which each of the units or composing modules tests a program independently. A unit test usually is done by programmers during system development. That means a programmer who writes a module for a system is responsible for testing that module, and there is no need for postponing the testing of that module until after system completion. Unit testing aims to determine the accuracy of the function of units that will be used after development in different parts of the system. Unit testing is usually considered part of white-box tests, which requires access to the inner structure of the testing code.

The integrity test aims at the reliability that different parts of the system are working well with each other and interactions, connections, data replacing are made accurately among different modules of a system, and therefore, the whole system has a correct function. Integrity tests can be performed at different levels. For example, we can consider each of the basic modules of the system as a system (it is composed of smaller components) and perform an integrity test. Moreover, we can consider the whole system as a unit system and test it. The notable point is that it should not be assumed that performing a unit test on system modules do not require performing the integrity test. Both types of mentioned tests are required, and each has its unique ability. The notable point in the integrity test is the contact points and interaction of modules with each other which test

the modules along with each other and their function, while the unit test considers modules independently and separately from other components of the system [2].

A system test is a fully integrated system to study whether all requirements are met. Before supplying the final copy of the software, alpha and beta tests are also performed. The acceptance test is performed based on documented requirements of system users, and it aims at acquiring the confidence of supplying all requirements of users by system. In other words, in this test, we want to be sure that the produced system is acceptable in users' view, so it is better to perform the test by users or their representatives in actual conditions and situations. One of the most challenging expectations in software testing is designing a good test case which is a complicated art. The test case is one of the essential subjects in software engineering, which have diverse definitions.

A test case summarizes the status of cases to be tested. For example, if testing is performed in a numerical field, one idea for this test will be entering letter characters in the field. It is an idea of whether a data group creates an error case. So, we use test case which includes particular inputs or unique methods for software testing. In this regard, in this research, we will use Imperialist Competitive Algorithm for producing test cases that have the most program covering. This algorithm is inspired by the social behavior of humans and has more intelligence than other algorithms inspired by other living creatures, and can be used as an efficient algorithm.

2. Related Works

The complexity of software systems in the last decade has been increased remarkably, and software testing as a compressed work has become ever-increasingly expensive; therefore, the technique which leads to automatic production test data will have the biggest potential for remarkable reduction of costs. Many experiences and researches are reviewed for studying test cases and solving problems during research.

The studies were done by Suri et al. [3, 4] compare the compound method of genetic algorithm and bee colony with ant colony algorithm for selecting test case in regression test and showed that combining genetic and bee colony algorithm is quicker than ACO (ant colony algorithm) and has a significant effect in reducing costs and execution time. In another research [4], Bhasin Et al. review the literature to find the gap in their proposed method. In their method, the cost of executing test cases is considered a basic regression test and is prioritizing criteria and using a genetic algorithm to reach the desired result that minimizes the test set. In 2010, Singh et al. propose an optimal method of time limitation, which uses an ant colony optimizing algorithm. The results provide motivation considered in the next research [5], and its algorithm is implemented with a few changes in which ACO can create optimal test cases successfully.

Wung et al. [6] introduce an automatic approach for producing test cases based on specifying the usages with domain modeling, which were used for providing automatic ground for detecting scenarios and test inputs of the combination of processing natural languages with another method. In [7], Sapienz presents an intuitive approach for testing android programs with a multipurpose structure based on searching. This approach explores problem space and simultaneously finds optimal sequence following the increasing problem searching area and error report. In development, Sapienz uses the combination of random fuzzy and systematic and exploratory search. Most software programs that are developing include a graphic interface. Systematic testing of these programs requires modeling test cases as a set of interface events produced and executed on software. Researchers and scholars believe that different techniques such as model-based, observation/replay, and manual scripted should be studied in graphic interface testing. Nowadays, graphic interface testing is performed with advanced tools.

Nguyen et al. [8] introduce a new tool named Guitar, which exploits different testing techniques in the graphic interface. One of the important features of this tool is using additives from which the users can benefit according to their requirements. The other group of software programs that users nowadays consider is cell phone software. Cell phone systems always support models based on concurrent programming or events. These models can create many concurrent errors due to creating diverse threads which are not concurrent with each other apart from features such as high responding speed. Predicting and findings this group of errors is very complicated and time-consuming. Meanwhile, in [9], researchers develop a strategy for testing graphic interface functions based on an optimizing algorithm of simple particles poll. The proposed strategy is presented to produce an optimal test case using an event graph. Meanwhile, this strategy can manage and restore test cases. One of the important

features of the paper is presenting a strategy that tests the graphic interface functions without studying the program code.

In [10], a new technique named RacerDroid is introduced to find concurrent errors in android programs. This technique produces good test cases by actively controlling events, scheduling, and arranging threads. Security in computer systems has become one of the important concerns of developers. One of the conventional methods of security test is the function block diagram (FBD). FBD is a PLC programming language developed using a graphical data stream. Authors of [11] proposed a test coating approach and structural test based on FBD. This approach considers all important data stream paths and functions proper test (blocs). Panichella et al. [12] model the production of test cases as a multipurpose problem. The main reason for this modeling is the availability of several goals in testing each software, and the balance among test goals was also studied. Finally, a solution named DynaMOSA is proposed, a dynamic multipurpose arranging algorithm. This approach is for producing developed test cases. DynaMOSA is based on hierarchy dependency control, and in other research, Kalaee et al. [13] propose an approach for software testing to balance precision and processing time. These two goal functions are contrary to each other such that increasing precision increases the processing time. The paper's main goal is to increase precision and reduce processing time. The proposed approach is based on a binary decision-making graph diagram and particle pool algorithm [14]. In the particle pool algorithm, each particle is a possible response for optimizing producing test case. Then particles search in the problem area to find a response that optimizes the precision and processing time. Modeling with a graph reduces complexity, optimal managing of bulk data, and simplifies the problem.

In [15], the automatic producing technique of test case is studied in general, and this technique is divided into five groups: 1) structure test via symbolic processing, 2) model-based test, 3) compound test, 4) random and adjustment random test and 5) search-based test and each of these techniques was studied. One important research in error finding in Pravin [16] is an experimental study on error location and effective choosing of test cases via neural network. The neural network has several advantages compared to other models, like its ability to learn. The neural network can learn data rules with or without supervision by fiving a simple dataset. Pattern detection, system detection, intelligence control, cost, credit estimation, and the ability to reuse are neural networks applications that can help us find software bogs. Pravin studies Tarantula error detections, set community, set sharing, nearest neighbor, and transactions reason and showed that the Tarantula method outweigh the other four methods in error detection and is less costly than the other four methods. Pravin concludes that the RBF neural network outweigh other methods studied in [16] regarding time and error detection.

Pravin et al. [17] introduce an error detection algorithm for prioritizing test cases or choosing test cases with higher error detection. Pravin implements his proposed method in an open-source system like Webkit. The implementation results show that increasing test cases is an effective method and improves the time budget and the number of detected errors compared to other methods. In [18], software testing is introduced as the main process in the software development life cycle. In this regard, a method based on prioritizing test cases is presented. This prioritizing depends on code coverage. In [19], the GUI function test strategy is proposed via simple SSO optimization. SSO is used to produce an optimal test set via EIG (event-interaction graph), and finally, in [20], a method is presented to minimize the number of test cases in a conscious structural test. In this method, the primary test is optimized via CS (Coco search) and a combined approach and used mutation test to remove different breaks in testing software and filter test cases based on detected breakpoints.

Efficient units are ranked by utilizing the factor of competition among imperialists to attract each other's colonies. One advantage of proposed method is that, without solving any mathematical and complex solution approaches, all extreme and non-extreme units are ranked only by comparing the pairs [21].

3. Proposed Approach

This section deals with the proposed method for increasing error tolerance. One of the essential parts of error-tolerant systems is the internal variables of the program. By using these variables and following the values of these variables, we can follow the availability or unavailability of error based on these values, while the notable point in this regard is using input data in the program, which could cover more lines of program to guide variables' values in paths with the most coverage. These paths can detect sudden variables changes via these paths and determine the error location. In this regard, this research is composed of different parts.

According to raised methods, one of the weak points of these methods is focusing on output results, we could not see the potential problems during program execution, and we should only decide based on output. In the raised proposed method, we can use good test data for studying the procedure and obtain values of variables after breakpoints of the errors during the program. The procedure studying makes it possible to study errors during program execution. Several program samples, like benchmark, will be used in this regard. These benchmarks are c# programs. Benchmarks should be able to challenge the program variables to test more intermediate data in encountering types of circles and conditional orders with benchmarks.

The features of the proposed method:

- Independency of the proposed method on only the output system;
- Observing variables values in breakpoints during program execution.

Shortcomings of the proposed method:

- Detecting breakpoints in the program;
- Equipping program for testing data test.

In this regard, section 3-1 will consider the manner of program equipping, determining breakpoints for determining passed paths by variables, and section 3-2 will describe the algorithm by embedding an execution process control via test data with Imperialist Competitive Algorithm since this algorithm could not cover the errors accurately without valid data.

3.1. System of program equipping and determining breakpoints

Before describing this part, we will introduce some of the terms.

- Basic Block.

A Basic Block (BB) is a maximal set of instructions that are ordered and non-branch (except in the last instruction) or branch destinations (except in the first instruction) where the first instruction is execution and the last one is leaving [22].

- Abstract Basic Block.

A basic abstract Block does not contain any other adequate instructions except an unconditional jump instruction and has only one successor node and one predecessor node [24].

- Program Flow Graph.

Flow graph is based on the connection among program lines which are surveyed by input data in which each line of a program represents a token like $t=\{t_1,t_2,t_3,\dots,t_n\}$ in which n shows program lines. The connection among these lines is shown via edges $E=\{e_1,e_2,e_3,\dots,e_n\}$ each line of a program is executed after the other line is connected with an edge, and if it is not executed, two lines of no edge are considered among these lines.

- Branch sequence.

A program takes different paths depending on the input applied. A path, a sequence of basic blocks, is chosen depending on the outcome of the control instructions encountered at run-time [25]. We called the sequence of basic blocks, which will be executed at run-time, a branch sequence.

3.1.1. Breakpoints

Some programs are used as a benchmark for detecting breakpoints. First, the text of these programs is read via breakpoints detecting programs and tokens. Token means putting lines of a program in separate parts to decide based on these parts about breakpoints. Based on the created token, conditional order is extracted. When conditional order or circle is seen in a token, the system considers this token as the onset of a conditional pointy and will search for the final token. These operations will be done based on the number of open and closed braces in token after the start point. A sample of implementing this method is displayed in continuing. The sample of the below code is considered input for finding the breakpoint (fig.1).

```
1-if (x < > 0)          T={ s1: x=0; z=1; y=1;
2-  then if (y>1)            s2: x=1; z=2; y=1;
3-      then y =5;           s3: x=1; z=3; y=3;}
4-      else y = y – x;
5-  else z=x;
6-if (z>1 && y > 1)
7-  then z = z /x;
8-end
```

Fig.1. Input Code

In this program, tokens are the lines in a program. Each of these lines is put in a list, and a graph is created after detecting breakpoints. In this graph, breakpoints are seen well. Fig.2 illustrates the graph based on input data in the program.

Fig.2 shows the numbers within nodes, program lines, or tokens.

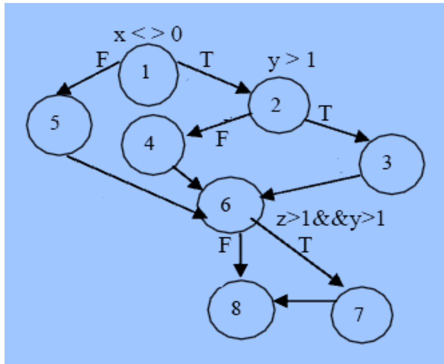

Fig.2. Program graph based on conditional orders

3.1.2 Equipping program for testing data

Program equipping means accessing variables' values, defined and initialized before breakpoints. The variables in every program are divided into two main groups:

- Input data: these data are used as program input, and their values are transferred to intermediate data for operations in the program. Intermediate data: these data are defined within the primary function during the program as used variables and are used during the program, and these data are called local variables.

Displaying the values of these variables after breakpoints is very important since we can observe and display the changes in variables within bloc (parts of the program which are put in the beginning and end of a breakpoint).

Displaying the values of these variables after breakpoints is very important since we can observe and display the changes in variables within bloc (parts of the program which are put in the beginning and end of a breakpoint). The raised system in this research automatically extracts the raised variables in the program, saves the values of these variables after breakpoints in a text file, and then displays them in the program. Program equipping comprises several parts, as seen in the flow chart below (fig. 3).

This flowchart uses the graph of program lines as a workflow graph. This graph can show the different program execution paths from the beginning to the end. These paths will help us a lot in finding breakpoints and simulating variables.

3.1.3 Imperialist Competitive Algorithm for producing test data

Imperialist Competitive Algorithm is a method in evolutionary computations which deals with finding the optimal response to different optimizing problems.

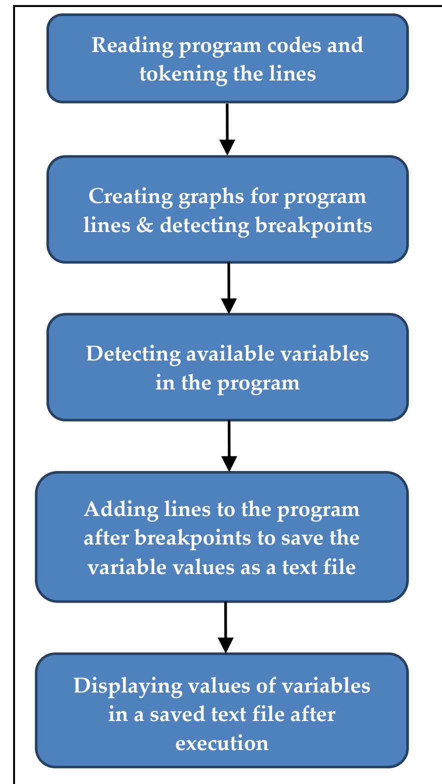

Fig.3. Flowchart of program equipping

This algorithm provides an algorithm for solving optimizing mathematical problems via mathematically modeling the social-political evolutionary process. The essential bases of this algorithm are composed of assimilation, imperialist competition, and revolution. This algorithm imitates countries' social, economic, and political evolutionary processes, and upon mathematically modeling parts of this procedure, it provides operators regularly as a country. Imperialist Competitive Algorithm stages will be studied in continuing. In optimization, the goal is finding an optimal response based on problem variables; we create an array of problem variables that should be optimized. For starting algorithm, we create N_country of the primary country to choose N_imp of the best members of this society (the countries with a minor cost function) as imperialist. The remaining N_col of countries composes colonies, each of which belongs to an imperial. To divide primary colonies among imperialists, we give several imperialist colonies proportional to their power. In this research, the countries show input variables for each benchmark. The below table 1 displays a country for a benchmark with six input variables. Each variable ranges randomly between -1000 and 1000.

Table 1. A country for a benchmark with six input variables

Value 1	Value 2	Value 3	Value 4	Value 5	Value 6
-10	25	100	-50	20	25

In this table, the number of variables is considered at 6.

In this algorithm, the fitness function is the number of passed lines by input data, and if the number of these lines is high, these values are accepted as test data. In this regard, stages of the Imperialist Competitive Algorithm are described as follows:

1. Choose the countries based on input data and create primary imperials;
2. Move the colonies toward imperialist countries (assimilation or absorption policy);
3. Produce a new country and replace it if the fitness function is better;
4. Apply revolution operator;
5. If there is a colony in an imperial which has less cost than the imperialist, replace the colony and imperialist;
6. Calculate the total cost of this imperial (considering imperialist and its colonies cost);
7. Choose one or more colonies of the weakest imperial and give it to the imperial which has the most probability of takeover;
8. Remove weak imperials;
9. If only one imperial remains, stop; otherwise, go to stage 2.

We can create test data to cover errors in programs based on a raised fitness function according to these variables.

4. Experimental Results

In order to assess the effectiveness of the proposed approach, five benchmark programs are chosen for the experiment: Quick Sort (QS), Matrix Multiplication (MM), Bubble Sort (BS), Linked List (LL), and Binary Search Tree (BST) while performance overhead, memory size overhead, error detection latency, and error detection coverage are obligatory parameters for evaluating our approach that should be measured and reported. The procedure is such that having evaluated the memory overhead and the performance loss results of the presented scheme, the average error detection latency of the presented scheme is analyzed, which follows with allotting to the error detection coverage. We adopted this method in which the faults are injected into the program by modifying the assembly codes of the source file. The assembly codes randomly applied one branch deletion, branch creation, or branch operand change. We consider four versions for each benchmark:

- The original code;
- A safe one obtained by applying the CFCSS [25] technique to the original code;
- A safe one obtained by applying the RSCFC [26] technique to the original code;
- A safe one obtained by applying the CFCBS technique to the original code;
- A safe one, obtained by applying our technique to the original code.

Table 1 shows the results during fault injection experiments. Fault effects are classified as follows:

- Wrong Result (WR): the fault modifies the program's results without being detected;
- Two thousand errors, including branch deletion faults, branch creation faults, or branch operand change faults, are injected for each assembly program; having compiled the faulty assembly program, a machine code is generated.

In order to assess the effectiveness of the proposed approach, five benchmark programs are chosen for the experiment: Quick Sort (QS), Matrix Multiplication (MM), Bubble Sort (BS), Linked List (LL), and Binary Search Tree (BST) while performance overhead, memory size overhead, error detection latency, and error detection coverage are obligatory parameters for evaluating our approach that should be measured and reported. The procedure is such that having evaluated the memory overhead and the performance loss results of the presented scheme, the average error detection latency of the presented scheme is analyzed, which follows with allotting to the error detection coverage. We adopted this method in which the faults are injected into the program by modifying the assembly codes of the source file. The assembly codes randomly applied one branch deletion, branch creation, or branch operand change. We consider four versions for each benchmark:

- The original code;
- A safe one obtained by applying the CFCSS [25] technique to the original code;
- A safe one obtained by applying the RSCFC [26] technique to the original code;
- A safe one obtained by applying the CFCBS technique to the original code;
- A safe one, obtained by applying our

technique to the original code.

Table 1 shows the results during fault injection experiments. Fault effects are classified as follows:

- Wrong Result (WR): the fault modifies the program's results without being detected;
- Two thousand errors, including branch deletion faults, branch creation faults, or branch operand change faults, are injected for each assembly program; having compiled the faulty assembly program, a machine code is generated.

Table 1.A. Experiment results of branch deletion and branch change faults injection into programs

Program	DEL (wr)(%)	Change (wr)(%)	Insert (wr)(%)
QS-original	36.7	25.4	28.7
QS-original	40.3	25.3	25.3
BS-original	48.7	26.7	16.7
LL-original	54.4	29.4	19.4
BST-original	40.9	30.9	26.5
QS-CFCSS	17.4	10.5	7.46
MM-CFCSS	12.7	5.30	14.5
BS-CFCSS	6.73	16.9	8.64
LL-CFCSS	10.42	18.2	7.20
BST-CFCSS	20.3	13.4	11.6
QS-RSCFC	15.1	7.82	6.20
MM-RSCFC	7.60	4.83	13.3
BS-RSCFC	5.34	13.2	4.24
LL-RSCFC	13.6	14.5	7.26
BST-RSCFC	15.4	11.6	8.64
QS-CFCBS	15.5	8.34	4.75
MM-CFCBS	8.92	5.35	12.9
BS-CFCBS	4.20	14.5	4.64
LL-CFCBS	11.80	16.1	9.68
BST-CFCBS	14.9	10.8	6.78
QS-our technique	14.9	6.87	4.36
MM- our technique	7.26	4.31	11.25
BS- our technique	3.69	13.21	4.52
LL- our technique	9.86	14.42	7.21
BST- our technique	14.2	10.6	6.31

Original programs and the hardened programs with CFCSS, RSCFC, CFCBS, and our technique under each fault type. As can be seen in Fig. 4, 5, 6. Thus, our technique can be compared with the best previous techniques in terms of fault coverage.

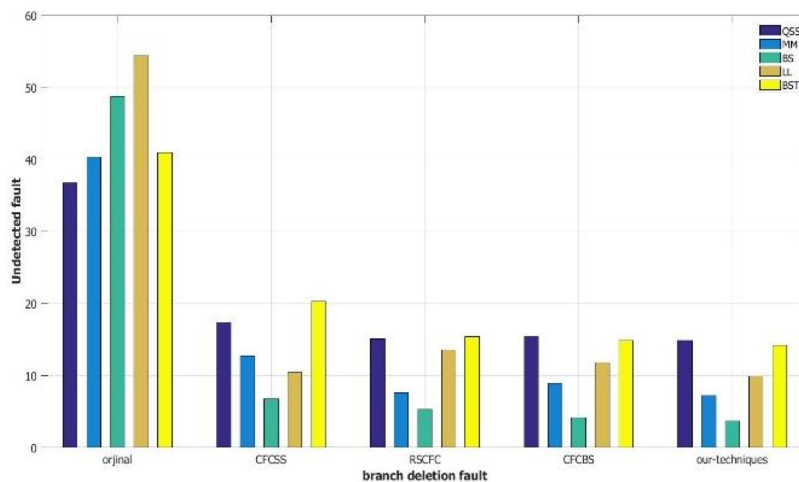

Fig. 4. The comparison of undetected faults branch deletion faults injection into programs

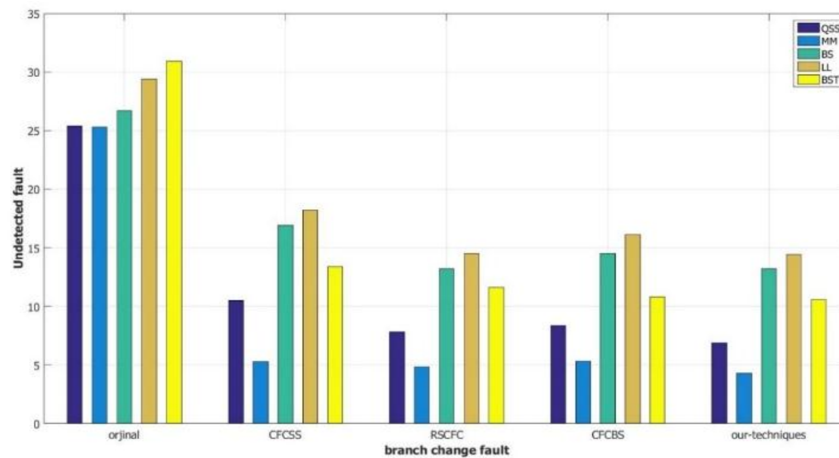

Fig.5. The comparison of undetected faults branch change faults injection into programs

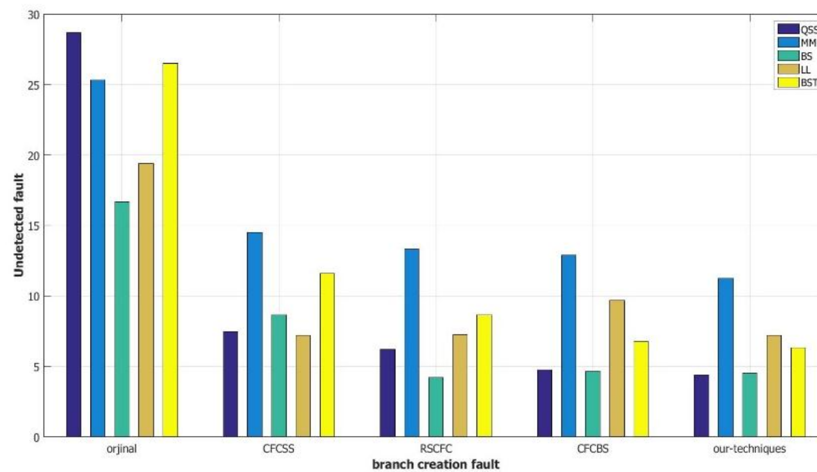

Fig.6. The comparison of undetected faults branch creation faults injection into programs

5. Conclusion

This paper presented a method for controlling and detecting error. The method was based on detecting breakpoints and paths by producing test data via Imperialist Competitive Algorithm for detecting error. In this regard, the program was initially equipped, and breakpoints were detected in the program. Having used error injection experiments on SPEC criteria, it was shown that the proposed method performed higher efficiency compared to previous techniques.

References

1. Parkin, R., & Australia, I. (1997) Software unit testing. The Independent Software Testing Specialists, IV & V Australia.
2. Häser, F., Felderer, M., Breu, R. (2014) Software paradigms, assessment types and non-functional requirements in model-based integration testing: a systematic literature review. In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (p. 29). ACM.
3. Suri, B., Mangal, I. (2012) Analyzing test case selection using a proposed hybrid technique based on BCO and genetic algorithm and comparison with ACO. International Journal of Advanced Research in Computer Science and Software Engineering, 2(4).
4. Bhasin, H. (2013) A Novel Approach to Cost Cognizant Regression Testing. International Journal of Computer Science and Management Research, 2(5), 2595-2600.
5. Suri, B., & Singhal, S. (2011) Analyzing test case selection & prioritization using ACO. ACM SIGSOFT Software Engineering Notes, 36(6), 1-5.
6. Wang, C., Pastore, F., Goknil, A., Briand, L., Iqbal, Z. (2015) Automatic generation of system test cases from use case specifications. In Proceedings of the 2015 International Symposium on Software Testing and Analysis (pp. 385-396). ACM.
7. Mao, K., Harman, M., Y. (2016) Sapienz: Multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (pp. 94-105). ACM.
8. Nguyen, B. N., Robbins, B., Banerjee, I., Memon, A. (2014) GUITAR: an innovative tool for automated testing of GUI-driven software. Automated software engineering, 21(1), 65-105.
9. Ahmed, B. S., Sahib, M. A., Potrus, M. Y. (2014) Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI

functional testing. Engineering Science and Technology, an International Journal, 17(4), 218-226.
10. Tang, H., Wu, G., Wei, J., Zhong, H. (2016) Generating test cases to expose concurrency bugs in Android applications. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (pp. 648-653). ACM.
11. Wu, Y. C., & Fan, C. F. (2014) Automatic test case generation for structural testing of function block diagrams. Information and Software Technology, 56(10), 1360-1376.
12. Panichella, A., Kifetew, F. M., Tonella, P. (2018) Automated test case generation as a many-objective optimization problem with a dynamic selection of the targets. IEEE Transactions on Software Engineering, 44(2), 122-158.
13. Kalaee, A., Rafe, V. (2016) An optimal solution for test case generation using ROBDD graph and PSO algorithm. Quality and Reliability Engineering International, 32(7), 2263-2279.
14. Jordehi, A. R. (2015) Enhanced leader PSO (ELPSO): a new PSO variant for solving global optimization problems. Applied Soft Computing, 26, 401-417.
15. Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W, Li, J. J. (2013) An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 86(8), 1978-2001.
16. Pravin, A., Srinivasan, D. S. (2013) An efficient algorithm for reducing the test cases is used for performing regression testing. In 2nd International Conference on Computational Techniques and Artificial Intelligence, Dubai (UAE) (pp. 194-197).
17. Pravin, A., Srinivasan, S. (2013) Effective Test Case Selection and Prioritization in Regression Testing. Journal of Computer Science 9 (5): 654-659, 2013 ISSN 1549-3636.
18. Ahmed, B. S., Sahib, M. A. & Potrus, M. Y. (2014) Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing. Engineering Science and Technology, an International Journal, 17(4), 218-226.
19. Ahmed, B. S. (2016) Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing. Engineering Science and Technology, an International Journal, 19(2), 737-753.
20. Alkhalifa, Z., Nair, V. S., Krishnamurthy, N., Abraham, J. A. (1999) Design and evaluation of system-level checks for on-line control flow error detection. IEEE Transactions on Parallel and Distributed Systems, 10(6), 627-641.
21. Hasan, B. K., Mohsen, R. M., Farhad, H., L. (2021) Ranking of decision making units using the imperialist competitive algorithm in DEA. Measurement and Control, 54(9), 1326-1335. https://doi.org/10.1177/00202940211028883
22. Jian-Li, L. I., Qing-Ping, T. A. N., Tan, L. F., Jian-Jun, X. U. (2014) Control flow checking Method based on Abstract Basic Block and Formatted Signature. Chinese Journal of Computers, 37(11).
23. Venkatasubramanian, R., Hayes, J. P., & Murray, B. T. (2003) Low-cost on-line fault detection using control flow assertions. In 9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003. (pp. 137-143). IEEE.
24. Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002) Control-flow checking by software signatures. IEEE transactions on Reliability, 51(1), 111-122.
25. Li, A., Hong, B. (2010) On-line control flow error detection using relationship signatures among basic blocks. Computers & electrical engineering, 36(1), 132-141.
26. Liu, L., Ci, L., Liu, W. (2016) Control-Flow Checking Using Branch Sequence Signatures. In 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (pp. 839-845). IEEE.