Regression Testing based on Hamming Distance and Code Coverage

Syed Akib Anwar Hridoy
North South University
Dhaka-1229, Bangladesh

Faysal Ahmed
North South University
Dhaka-1229, Bangladesh

Md. Shazzad Hosain
North South University
Dhaka-1229, Bangladesh

ABSTRACT

A software testing process that tries to uncover new bugs for an existing system from the previous test suite due to expansion of the software is known as Regression Testing. The test suite will hold the same test cases that were tested for the system in its earlier version. For regression testing, prioritizing the test cases is always a complex as well as challenging task. In fact researchers have been proposing many approaches to arrange the test cases so that the cost of the software can be reduced in terms of human labor, time, and money as well. Many such approaches have shown quite good results too. In this paper, we have proposed a new approach of prioritizing the test cases that extends hamming distance based prioritization with code coverage based techniques. Our proposed method helps to unfold the previous bugs as well as the newly arrived bugs at the early cycle of the regression testing.

Keywords:

Regression Testing, Test Case Prioritization, Hamming Distance, Code Coverage

1. INTRODUCTION

We cannot deny the fact that a software may expand or patch a specific module in its life-cycle. Every time whether we introduce new code or fix the existing software code several test cases are introduced in the test suite to validate the new features or the patch. To build confidence on software we are not only required to test software with 'new' test cases but also to perform regression testing [1]. The purpose of regression testing is to run all the previous test cases to ensure that all the functions that were part of the software before the update still work fine. As new test cases are added to the existing test suite, it tends to grow larger often making it too costly to execute entire suite. Software testers cannot ignore the 'new' test cases that are designed for the added features, but they want to spend less amount of time and energy for earlier test cases to make the software cost-effective. Thus, a natural goal is to run minimum number of previous test suits but to maximize the test objectives. To achieve the goal researchers studied different approaches such as *minimization*, *selection* and *prioritization* to maximize the value of the accrued test suite [2]. While *minimization* eliminates redundant test cases in order to reduce the number of tests to run, *selection* seeks to identify the most relevant test cases for the recent changes. Once the number of test cases is lowered, either by minimization or by selection, *prioritization* may be applied to order test cases so that faults are detected at early stages of testing.

In the literature we found a great deal of research work on different regression testing approaches, some focused on test selection [3, 4], some on minimization [5] and others on prioritization [6, 1]. However, quite often two or more approaches [7, 8] were combined for better results. The approaches also differ in emphasizing different goals to achieve, such as code coverage maximization [9], requirement based prioritization [10], software failure severity, cost cognizant test prioritization [1] and many more. In our research, we combine two approaches of prioritization and minimization, first to prioritize the selected test cases for early fault detection and then to minimize the number of test cases from the accrued test suits to achieve 100% code coverage.

The goal of regression testing is to rerun previous test cases and to check whether program behavior has changed as well as whether previously fixed faults have re-emerged. A common technique is to prioritize and rerun test cases that identified faults previously and to check whether the faults have re-emerged. However, it does not ensure 100% code coverage, which is important to check whether the program behavior has changed or not. Thus, many researchers suggested code coverage based prioritization techniques [9, 11] that achieve total code coverage at the earliest. But, code coverage based techniques do not always detect more bugs at the earliest. To overcome this, other techniques such as hamming distance based prioritization [6, 12] were studied, which on the contrary ignored 100% code coverage. In this paper, we thus extend hamming distance based prioritization with code coverage technique to achieve both the goals of regression testing. Finally, we have compared our proposed approach with the hamming distance based approach and showed the improvement.

The paper is organized as the following. Section 2 presents problem formulation, section 3 describes code coverage based prioritization, section 4 explains hamming distance based prioritization, section 5 illustrates our proposed approach and its analysis, section 6 provides literature review and finally section 7 draws the conclusion.

2. PROBLEM FORMULATION

Let P be a program, P' be a modified version of P, T be a test suite developed for P. Regression testing is concerned with validating P'. The main objective of Software Quality Assurance (SQA) engineers is to complete regression testing to achieve its goals within a short time so that software cost is lowered. To reduce

the cost of regression testing, firstly test cases are reduced and then prioritized to find out bugs at early stages.
Rehman et al. [9] defines test case reduction problem as the following:

DEFINITION 2.1 TEST CASE REDUCTION. *Given a test suite T_N with N number of test cases, the reduction problem is to retain $T_M \subset T_N$ with M number of test cases.*

Bushra et al. [1] defines the test case prioritization problem as the following:

DEFINITION 2.2 TEST CASE PRIORITIZATION. *Given T, a test suite; T_p, the set of permutations of T; f, a function from T_p to the real numbers; the problem is to find $T' \in T_p$ such that for all $T'' \in T_p, T' \neq T'', [f(T') \geq f(T'')]$.*

Here, T_p represents the set of all possible prioritization (orderings) of T and f is a function applied to any such ordering, yields an award value for that ordering [1].
Now, let us consider a flow graph of a program shown in Fig. 1a that has two faults F_1 and F_2. While testing the program path *A-B-C-D-H* and *A-C-D-H* expose these two faults. Now, assume that fault F_3 has arisen, as shown in figure 1b, while debugging and fixing faults of F_1 and F_2.

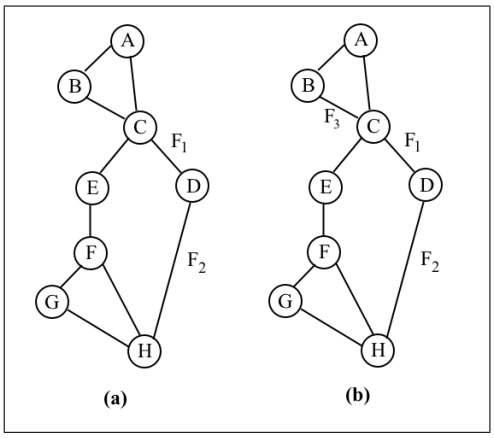

Fig. 1. Flow graph of a test program

According to hamming distance based prioritization proposed in [6, 12], one of the two paths would be selected randomly. If path *A-C-D-H* is chosen randomly then the fault F_3 goes unnoticed. If we could somehow choose the path *A-B-C-D-H* instead of *A-C-D-H*, then all the faults F_1, F_2, F_3 would be detected at once. On the other hand, code coverage based prioritization [9, 11] would choose the path *A-B-C-E-F-H* as it covers more nodes or path that only uncovers F_3. In this case, the path *A-C-D-H* that uncovers the faults F_1 and F_2 go below the list. Code coverage based prioritization makes sure that all the paths are covered. But it takes higher number of test cases to be executed to cover 100% path.
In our research, we have combined both hamming distance and code coverage based techniques to detect previous faults earlier as well to achieve 100% code coverage to detect the newly cropped up faults.

3. CODE COVERAGE BASED PRIORITIZATION

Code coverage is a way of measuring how many statements or blocks or lines are executed when a test case has run. A program which is thoroughly tested with 100% code coverage has a lower chance of having bugs. Test cases are prioritized to cover most of the path in early stage of testing [13].
For figure 1b, let us explain the code coverage based prioritization technique according to [13].

Table 1. Test Cases and Statement Coverage

Test Case#	**Statements**							
	A	B	C	D	E	F	G	H
T_1	1	0	1	1	0	0	0	1
T_2	1	1	1	1	0	0	0	1
T_3	1	1	1	0	1	1	0	1
T_4	1	1	1	0	1	1	1	1
T_5	1	0	1	0	1	1	0	1
T_6	1	0	1	0	1	1	1	1

Table 1 is created from the figure 1b. Each node or path is considered as a separate statement and test cases are created to cover all the paths. If any particular test case covers any statement, then a value '1'is assigned for that column in the table. The test case row that has the highest sum among all of these test cases is selected first into the test case execution set and the statements (columns) are removed from the table to select the next test case. This procedure continues until all the statements are removed from the table.
In this specific example, the procedure yields a set $T = \{T_4, T_1\}$ to be executed for regression testing.
In the execution test set of code coverage based prioritization, T_4 has higher prioritization as it covers most path although T_1 detects higher number of bugs than T_4. This clearly indicates that covering more paths does not always find out bugs early.

4. HAMMING DISTANCE BASED PRIORITIZATION

Hamming distance [14] was originally proposed to compute the number of different bits in two sequences. For example, "1010" and "1001" are at a distance of 2, because two bits differ. It can also be adapted for two strings e.g. "Karolin" and "Kerstin" of same length that differ in three characters and have a distance of 3.
Maheswari et. al. [6] performs the hamming distance calculation to get an ordered list of test cases which lead the system to detect all the bugs. They start with the test case which has the most number of 1's. When two or more test cases have an equal number of bugs, they randomly choose a test case among those. Then another test case is selected to perform the OR operation with the first selected test case. After the OR operation, another test case is selected based on the most hamming distance to perform the OR operation with the result again. This process continues until all the bugs are unveiled. Let us explain the process through the following example.
Table 2 shows a sub-set of data from [15], provided by University of California Riverside, which is also used by [16], [10]. The column of the table is equal to the number of mutant version of the software we are considering and the row is the number of test cases we are considering for the experiment. The value of each cell is 1 if the test case can identify the fault, otherwise cell is filled up with 0. Code coverage for each test cases is calculated by C++ Coverage Validator [17].
Table 2 shows that T_2, T_4, T_5, T_6, T_7, T_9, T_{10}, T_{12} has the most numbers of 1's. T_2 is chosen randomly to start the calculation. And

Table 2. Results And Code Coverage of Mutant Codes

Test Case#	Mutant Codes						CC(%)
	F_1	F_2	F_3	F_4	F_5	F_6	
T_1	0	0	0	0	0	0	70.23%
T_2	0	0	1	1	0	0	10.53%
T_3	0	0	0	0	0	1	75.00%
T_4	1	0	0	0	1	0	77.37%
T_5	0	0	1	1	0	0	72.37%
T_6	1	0	0	0	1	0	72.37%
T_7	0	0	1	1	0	0	81.58%
T_8	0	0	1	0	0	0	75.00%
T_9	0	1	1	0	0	0	69.28%
T_{10}	1	0	0	0	1	0	72.37%
T_{11}	0	0	1	0	0	0	75.00%
T_{12}	0	1	1	0	0	0	77.63%
T_{13}	0	0	1	0	0	0	75.00%

CC(%): Code Coverage(%)

to perform the OR operation, we need another test case which has the most hamming distance with T_2.

As we know that hamming distance between two strings can be calculated from the number of corresponding mismatches bits, the distance between T_2 and T_1 is 2 as it mismatches in F_3 and F_4 columns. Accordingly we calculate other test cases hamming distances from T_2. We have found out that the hamming distance of T_4 from T_2 is 4 and T_6, T_{10} also get the distance of 4 which is higher than any other non selected test cases. So, T_6 is chosen at random to perform the OR operation. Now, T_3 has the highest hamming distance with the output. So, OR operation will be performed between them. Then, T_9 and T_{12} make tie and this time T_9 is chosen at random. After the OR operation, we get output with all 1's. Figure 2 illustrates the whole procedure.

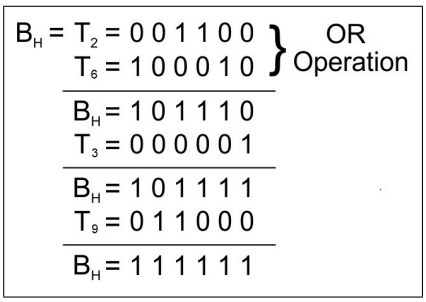

Fig. 2. Hamming Sequence Calculation.

So the prioritization according to [6] is,

$$T_2 \rightarrow T_6 \rightarrow T_3 \rightarrow T_9$$

5. PROPOSED APPROACH AND ANALYSIS

Section 4 shows that the hamming sequence can uncover most number of bugs with the least number of test cases at the early stages of testing. Our proposed method is thus more likely to section 4, but we are also considering the code coverage information to prioritize the test cases and ensure that all the paths of the program are covered.

While hamming distance based approach chooses randomly from two or more test cases that unveils the same number of faults, our proposed approach analyzes their past code coverage data and select the test case that covers more code in its execution than the others. For each of the test cases we have code coverage (CC%) information from the last build of the software. From this binary matrix we can find out the test cases from hamming distance calculation which can uncover all the faults. The point to be noted that no test case is selected at random when there is a tie between two or more test cases with the output result. Still there will be some paths that will not be covered as hamming distance based approach do not consider 100% code coverage. To overcome this problem, we will include some test cases in our prioritization list which help us to cover 100% path. Let us explain our proposed approach by an example.

As of Section. 4, we start the calculation with the test case which detects the most number of bugs. From table 2, we can see that T_2, T_4, T_5, T_6, T_7, T_9, T_{10} and T_{12} identify most number of bugs. According to hamming distance based approach T_2 is chosen at random, but we select T_7 as it has the highest code coverage among those test cases. To perform the OR operation, we require another test case which has the highest hamming distance with T_7. In this case, T_4, T_6 and T_{10} has equal number of hamming distance with respect to T_7. As T_4 has higher code coverage among them, it will be chosen. After the OR operation, T_3 get the maximum distance with the output. After performing the OR operation, T_9 and T_{12} has the maximum distance with the output. We take T_{12} instead of T_9 as T_{12} has higher code coverage than T_9. Figure 3 illustrates the calculation of proposed sequence.

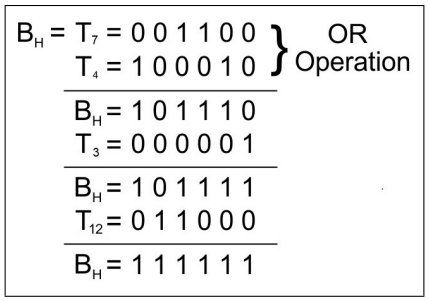

Fig. 3. Extended Hamming Sequence Calculation.

So, extending hamming distance based approach with code coverage information gives us the following prioritized sequence:

$$T_7 \rightarrow T_4 \rightarrow T_3 \rightarrow T_{12}$$

which is different from the hamming distance based prioritization order

$$T_2 \rightarrow T_6 \rightarrow T_3 \rightarrow T_9$$

The new test order covers more code than the previous hamming distance based order as shown in figure 4 and figure 5. The reason to cover more code was to detect new bugs at the earliest and our approach also starts with higher code coverage than hamming sequence from the very first execution. To justify our approach, we have now introduced an extended version of table 2 with two more faulty version of the program on table 3.

By the hamming distance based prioritization order, the newly arisen bugs remain hidden. While the extended version could reveal F_7 though F_8 is not revealed at this stage. This is because the extended version did not achieve 100% code coverage as shown in figure 5 and the bug F_8 remains in untested path. To overcome

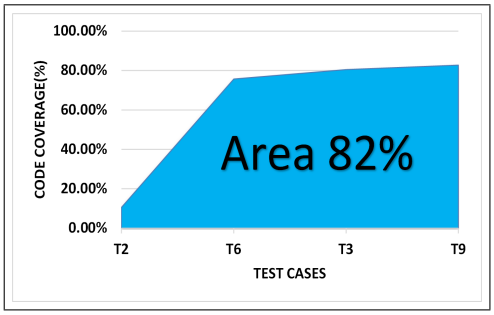

Fig. 4. Code Coverage of Hamming Distance Based Prioritization

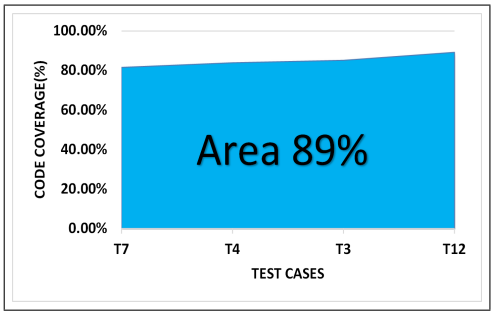

Fig. 5. Code Coverage of Extended Hamming Distance Based Prioritization

Table 3. Results and Code Coverage of Mutant Codes

Test Case#	**Mutant Codes**								**CC(%)**
	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	
T_1	0	0	0	0	0	0	0	0	70.23%
T_2	0	0	1	1	0	0	0	0	10.53%
T_3	0	0	0	0	0	1	0	0	75.00%
T_4	1	0	0	0	1	0	0	0	77.37%
T_5	0	0	1	1	0	0	0	0	72.37%
T_6	1	0	0	0	1	0	0	0	72.37%
T_7	0	0	1	1	0	0	1	0	81.58%
T_8	0	0	1	0	0	0	0	0	75.00%
T_9	0	1	1	0	0	0	0	0	69.28%
T_{10}	1	0	0	0	1	0	0	0	72.37%
T_{11}	0	0	1	0	0	0	0	0	75.00%
T_{12}	0	1	1	0	0	0	0	0	77.63%
T_{13}	0	0	1	0	0	0	0	1	75.00%

CC(%): Code Coverage(%)

this, finally we include T_{13} and T_{10} to cover 100% code and the new test order is the following.

$$T_7 \rightarrow T_4 \rightarrow T_3 \rightarrow T_{12} \rightarrow T_{13} \rightarrow T_{10}$$

Figure 6 shows that the new test cases achieve 100% code coverage. Also, considering the new bugs, our proposed ordering achieves higher bug detection ratio ρ as shown in figure 7. The higher bug detection ratio ρ is defined as the following:

$$\rho = \frac{N_d}{N_T} \times 100\% \quad (1)$$

where,

ρ = percentage of bug detection by test case T_i

N_d = number of bugs detected by test case T_i
N_T = total number of bugs

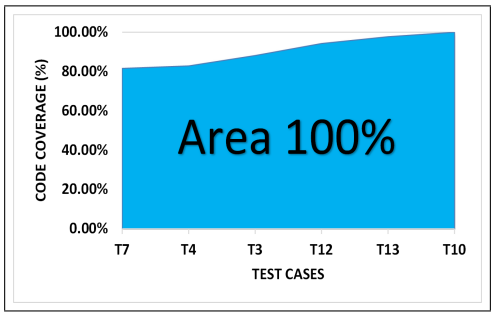

Fig. 6. Code Coverage of Proposed Sequence.

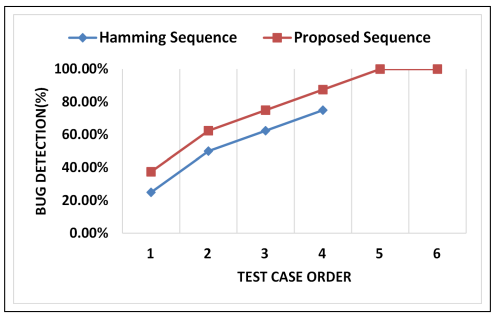

Fig. 7. Bug Detection Comparison.

Figure 7 clearly shows that our proposed technique has started with more bug detection rate for table 3 than the hamming sequence based technique. For this specific problem, hamming sequence based prioritization can only detect 75% bugs whereas our approach ends up finding 100% bugs. Our proposed approach thus ensures higher bug detection rate.

6. LITERATURE REVIEW

R. Kavitha et al. [10] suggested to order the test cases based on the requirements. Customer assigned priority, code implementation complexity and requirement change were considered as factors. Scoring policy for factors ranges from 0-10 are determined by clients and developers. Averaging the sum of these 3 scores decide the order of the test case.

Zeng and Wang [16] proposed a generic approach. They defined a multi-dimensional model which makes the prioritization model more flexible. They took consideration of all the dimensions that are more related to the project cost which are code coverage, fault exposing prioritization (FEP), requirement properties, historical information and execution time. They bound these dimensions into an equation. The output from the equation decides which test cases will be executed earlier.

Wenhong Liu et al. in [18], presented methods which modify existing test case prioritization algorithms in order to obtain prioritized test cases. Software requirement prioritization, software failure severity and probability rates were the factors for prioritization. Initially prioritized test cases were generated and then adjusted, or re-ranked based on the adjustments. Inclusion

of failure probability and severity improved the performance noticeably as seen in the experimental results.
In [19], the existing approaches have two different phases; test case prioritization and test case execution. They build a framework which could relate these two phases. They propose an adaptive approach which schedules the test cases in run time. The framework picks a test case from the test suite, run it and replace the output of the previous version for that test case and calculate the fault-detection capability of the not selected test cases based on the information available from the previous version and then select another test case with the largest fault-detection capability. This process goes on until the whole test suite is prioritized.
Ramaraj et al. [3] proposed a new concept in Regression Testing named 'Agent Based Regression Testing'. A dynamic approach for removing the Complexity of prioritizing test cases are done by monitoring code changes and generating test cases for the changed version only.

7. CONCLUSION AND FUTURE WORKS

In this paper we have introduced a regression testing approach that combines both the code coverage and hamming distance based techniques to detect previous faults and the newly arisen faults. Our approach covered more code from the beginning of its execution than hamming distance based prioritization and also ensures 100% code coverage. Results and analysis showed that this approach can unfold the previous bugs early and can look up for new bugs as well without changing the prioritization suite. Our research also helps software testers to find bugs in the premature stage of testing phase with least amount of test cases. It implies that our approach is cost efficient for both in terms of money and human resource.
The mutant code we have introduced in our research does not represent the actual bugs for a program. The experiment field was relatively small. This approach needs to be tested in large scale to see if the analysis holds.
In this paper, we only considered code coverage along with hamming distance. However, there are other parameters like execution time, customer requirement etc. that can be added to calculate a prioritized order. Allowing those approaches with variable distribution score formula along with hamming distance can make it work for all types of systems.

8. REFERENCES

[1] Bushra Hoq, Samia Jafrin, and Shazzad Hosain. Dependency cognizant test case prioritization. In *International Conference on Computational Intelligence and Software Engineering (CiSE 2011)*, Wuhan, China, December 2011.

[2] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[3] T.M.S.Ummu Salima, A. Askarunisha, and N. Ramaraj. Enhancing the efficiency of regression testing through intelligent agents. In *International Conference on Conference on Computational Intelligence and Multimedia Applications*, volume 1, pages 103–108, Dec 2007.

[4] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3):289–321, 2011.

[5] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Software Engineering Notes*, 31(1):35–42, 2005.

[6] R.U. Maheswari and D. JeyaMala. A novel approach for test case prioritization. In *IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pages 1–5, December 2013.

[7] Ruchika Malhotra, Arvinder Kaur, and Yogesh Singh. A regression test selection and prioritization technique. *Journal of Information Processing Systems*, 6(2):235–252, 2010.

[8] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 160–170, New York, NY, USA, 2014.

[9] S.U. Rehman Khan, Sai Peck Lee, R.M. Parizi, and M. Elahi. A code coverage-based test suite reduction and prioritization framework. In *4th World Congress on Information and Communication Technologies (WICT)*, pages 229–234, Dec 2014.

[10] R. Kavitha, V.R. Kavitha, and N.S. Kumar. Requirement based test case prioritization. In *IEEE International Conference on Communication Control and Computing Technologies (ICCCCT)*, pages 826–829, October 2010.

[11] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy. Code coverage-based regression test selection and prioritization in WebKit. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 46–55, Sept 2012.

[12] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

[13] R. Beena and S. Sarala. Code coverage based test case selection and prioritization. *CoRR*, 2013.

[14] Rw Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[15] University of California Riverside. Effectiveness of different test case prioritization methods based on coverage criteria.

[16] Xiaolin Wang and Hongwei Zeng. Dynamic test case prioritization based on multi-objective. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6, June 2014.

[17] Software Verification Limited. C++ coverage validator 64/32 bit evaluation.

[18] Wenhong Liu, Xin Wu, WeiXiang Zhang, and Yang Xu. The research of the test case prioritization algorithm for black box testing. In *5th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 37–40, June 2014.

[19] Dan Hao, Xu Zhao, and Lu Zhang. Adaptive test-case prioritization guided by output inspection. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 169–179, July 2013.