Total Coverage Based Regression Test Case Prioritization using Genetic Algorithm

Patipat Konsaard
Department of Computer Engineering
Chiang Mai University
Chiang Mai, Thailand
patipat_konsaard@cmu.ac.th

Lachana Ramingwong
Department of Computer Engineering
Chiang Mai University
Chiang Mai, Thailand
lachana@eng.cmu.ac.th

***Abstract*— Regression Testing is a test to ensure that a program that was changed is still working. Changes introduced to a software product often come with defects. Additional test cases are, this could reduce the main challenges of regression testing is test case prioritization. Time, effort and budget needed to retest the software. Former studies in test case prioritization confirm the benefits of prioritization techniques. Most prioritization techniques concern with choosing test cases based on their ability to cover more faults. Other techniques aim to maximize code coverage. Thus, the test cases selected should secure the total coverage to assure the adequacy of software testing. In this paper, we present an algorithm to prioritize test cases based on total coverage using a modified genetic algorithm. Its performance on the average percentage of condition covered and execution time are compared with five other approaches.**

Keywords—Test case prioritization, Test suite, Regression testing, Genetic algorith, Code coverage, APCC, Software Testing, Software engineering

I. INTRODUCTION

Software Testing is vital to software quality. It involves checking if software is adequate for its intended use. When a change is introduced to software, it is modified to reflect implementation of the change. Regression testing is, thus, required to test whether software still works as it did [1]. Regression testing is repeatedly performed throughout the software life cycle. However, regression testing can be costly and time consuming [2]. Consequently, various research efforts have been put through improving test case selection and prioritization [3][4][6][12].

Due to time and cost limitations of software development, retesting software using the entire suites is obviously expensive and inefficient. This shortcoming, thus, stimulates efforts to provide assistance to the regression testing process. The three main fields comprise test case selection, test suite minimization and test case prioritization [2]. Test case selection techniques [11] select a subset of test cases. Test case minimization techniques [4][10][11] reduce a test suite that still maintains the coverage. Test case prioritization [3][7] involves ordering test cases to have higher priority test cases executed earlier. Most work on test case prioritization techniques which focus on improving the rate of fault detection [13]. Another performance goal is to run test cases that achieve total code coverage in a timely manner [3][14][15]. Whereas detecting faults early is ideal to test case prioritization, identifying faults in advance is far more difficult. This paper, therefore, focuses on test case prioritization technique for total code coverage.

Several prioritization techniques are proposed [1][3][16][18]. A genetic algorithm is one of the heuristics widely used to solve problems in various domains including test case prioritization. The main benefit of a Genetic Algorithm technique is that it can be easily varied and customized to suit the problem. Additionally, genetic operators also contribute test case generation.

This paper proposes a test case prioritization technique that achieves 100% code coverage. Section II describes research background and related work. Section III introduces the research methodology. Section IV describes how the experiment is carried out and presents the results. Section V discusses the results. Section VI concludes the paper.

II. BACKGROUND AND RELAND WORK

A qualitative survey of regression testing practices by Engstrom and Runeson [1] reveals the increasing trend in providing support for regression test case selection, minimization and prioritization, especially. Challenges include providing automated support for regression testing techniques to promote practical implementation.

Kaur and Goyal [3] proposed a genetic algorithm for test case prioritization techniques using code coverage. The prioritized test cases give high value of APCC but not as high as the optimal order. The final test suite is responsible for achieving almost 90% code coverage. Interestingly, the control approaches which are Random Order and Reverse order gain higher APCC values than the GA order.

Ahmed, Shaheen, and Kosba [12] proposed a test suite prioritization technique using a Genetic Algorithm to increase test code coverage. In this work, multiple coverage criteria are taken into account to calculate fitness value. The outcome (APFD) of the proposed fitness function, however, was still,far from the optimal order. This can be implied that a single criteria fitness function may generate the better outcome.

978-1-4799-7961-5/15/$31.00 ©2015 IEEE

A new test case reduction technique called TestFilter was proposed by Khan and Nadeem [4]. Weight criteria are used in the reduction algorithm. Similar to other reduction techniques, TestFilter eliminates unnecessary test cases, and, hence, reduces related costs such as test case storage and execution costs. The effectiveness of test cases generated was not yet studied.

Athar and Ahmad [8] presented a test suite optimization technique using a Genetic Algorithm. The test suite formed gives 100% code coverage. As the chromosome size increases, the number of iterations required to get the optimized test suite is also increased. This kind of overhead could be reduced via preprocessing.

Srivastava and Kim [9] present a method to identify the most critical path clusters in a program using Genetic Algorithm with the main aim to increase software testing efficiency. The method is compared with a random test generation method. By identifying the most critical path, the rate of fault detection could be increased. However, fault detection cannot be guaranteed.

Nirpal and Kale [10] proposed a Genetic Algorithm to test case generation for path testing. The Genetic Algorithm is praised for its simplicity. The modification of their algorithm is required to get rid of unwanted paths and correctly locate the desired path. This indicates that some kind of preprocessing and/or post processing for Genetic Algorithms is needed to enhance the effectiveness of the technique employing Genetic Algorithms.

Fraser and Arcuri [11] proposed a novel technique implemented as a part of testing tool called EvoSuite to generate test suites with high coverage. This work focuses on branch coverage. The results show that smaller test suites give higher coverage.

Akarunisa and colleagues [7] proposed a metric for assessing the effectiveness of prioritization techniques in terms of fault detection rate and coverage. The new metric APFDc assesses the rate of fault detection by considering varying test cases and fault costs. Moreover, other new metrics are also presented including Average Percentage of Statement Coverage (APSC), Average Percentage of Branch Coverage (APBC), Average Percentage of Loop Coverage (APLC) and Average Percentage of Condition Coverage (APCC) based on the coverage criterion of various prioritization techniques.

The background research illustrates that Genetic Algorithms [19][20] are widely used for optimization. The simplicity of Genetic Algorithm makes it a suitable candidate to implement prioritization techniques. However, the Genetic Algorithm should be modified to achieve higher code coverage and reduce prioritization effort.

III. RESEARCH METHODOLOGY

This paper presents a test case prioritization technique using Genetic Algorithms. To illustrate how it works, the triangle problem is exercised since it is a well-known example in software testing education. The decision graph for this problem is derived for application of software testing. It is shown on figure 1. The graph illustrates the number of conditions covered by each path.

Table I presents test cases randomly generated for this problem. Test case related data such as inputs, corresponding outputs, paths, conditions and nodes covered are also shown on the table.

Next, the test suites are randomly generated based on the criteria that each of them satisfies the total coverage. Then, the fitness function is calculated. The population in top ranking of fitness function is selected as parents for crossover and mutation to create test cases that still satisfy the total coverage. Duplication is checked to make sure the final test suite meets the cost and time objectives of regression testing. Finally, the effectiveness of the Genetic Algorithm approach and the other five approaches are analyzed using the Average Percentage of Code Coverage (APCC) metric.

In this study, Bee Colony Optimization (BCO) is also studied to verify if a more complicated optimization algorithm will perform better than a simple algorithm like Genetic Algorithm. The BCO algorithm randomly discovers the initial population of test suites. Then, iteratively improves them and discards low quality solutions. The subsequent section explains in more detail how the modified Genetic Algorithm is performed.

IV. EXPERIMENT AND RESULTS

This experiment is done to study test case prioritization using Genetic Algorithm and its performance. The Genetic Algorithm is applied to the set of generated test cases. The experiment is organized in five steps: (1) graph generation, (2) test case generation, (3) test suite generation, (4) fitness calculation and (5) Genetic Algorithm application. The results obtained from applying the Genetic Algorithm is presented and discussed. The steps are explored in the following sections.

A. Experiment and result

The experiment is divided into five steps. Graph generation is done in the first step. In the beginning, a control flow graph for the selected problem is generated. Then, the decision graph (or decision-to-decision (DD) graph) is derived. It is used to determine the independent paths. As a result, the following paths are identified; ABDLM, ABCDLM, ABCEFGLM, ABCEFJLM, ABCEHJLM, ABCEHIJLM, ABCEHIKLM. Figure 1 presents the decision graph resulted from this step.

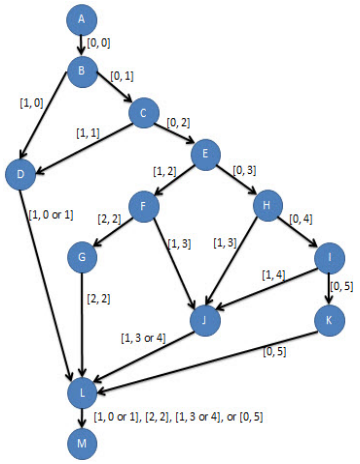

Fig. 1. A decision graph.

Next, the second step deals with generating test cases. In this experiment, the population size is set to 20. The test cases are randomly generated according to the population size. Table I shows the population created for the study and its related information such as data, expected output, independent path, condition and node covered by each test case.

TABLE I : GENERATED TEST CASES.

Test Case	X,Y,Z	Expected Output	Path	Condition Covered	Node Covered
T1	1,2,3	Not a Triangle	ABCDLM	2	1,2
T2	1,3,2	Not a Triangle	ABCDLM	2	1,2
T3	2,3,1	Not a Triangle	ABCDLM	2	1,2
T4	4,4,4	Equilateral	ABCEFGLM	4	1,2,3,4
T5	3,2,1	Not a Triangle	ABCDLM	2	1,2
T6	2,2,3	Isosceles	ABCEFJLM	4	1,2,3,4
T7	3,4,5	Scalene	ABCEHIKLM	5	1,2,3,5,6
T8	6,4,3	Scalene	ABCEHIKLM	5	1,2,3,5,6
T9	2,3,2	Isosceles	ABCEHIJLM	5	1,2,3,5,6
T10	1,2,4	Not a Triangle	ABCDLM	2	1,2
T11	1,4,2	Not a Triangle	ABCDLM	2	1,2
T12	2,1,4	Not a Triangle	ABCDLM	2	1,2
T13	3,6,4	Scalene	ABCEHIKLM	5	1,2,3,5,6
T14	6,3,4	Scalene	ABCEHIKLM	5	1,2,3,5,6
T15	0,0,0	Not a Triangle	ABDLM	1	1
T16	2,0,1	Not a Triangle	ABDLM	1	1
T17	4,2,2	Not a Triangle	ABCDLM	2	1,2
T18	4,6,3	Scalene	ABCEHIKLM	5	1,2,3,5,6
T19	3,2,2	Isosceles	ABCEHJLM	4	1,2,3,5
T20	4,3,6	Scalene	ABCEHIKLM	5	1,2,3,5,6

The third step deals with generating test suites to satisfy the number of conditions covered. Then, they are checked for duplication. If there is a duplicate, the algorithm randomly picks another test case number to replace the duplicated one. In addition, the chosen test case must also satisfy the number of conditions covered by the test suite. If the chosen test case results in a test suite that covers less conditions than the original test suite, the algorithm has to pick another test case. This process is repeated until all test cases in test suite are unique. The final test suite must fulfill the requirement of a candidate test suite by including the test cases that covers all conditions. This is to ensure that the test suite meets the coverage condition of quality software testing. Table II demonstrates the resulting test suites from the first iteration of these steps.

TABLE II : TEST CASES OBTAINED FROM THE FIRST ITERATION

Test Suites	Observation for first of Iteration										
TS1	T7	T6	T4	T9	T20	T15	T1	T3	T10	T19	T12
TS2	T19	T9	T6	T8	T4	T2	T18	T16	T5	T11	
TS3	T13	T4	T9	T15	T3	T6	T14	T19	T10	T17	
TS4	T5	T9	T4	T6	T7	T19	T16	T11	T8	T12	T17
TS5	T12	T13	T9	T19	T4	T15	T10	T6	T20	T1	
TS6	T6	T13	T19	T16	T10	T4	T9	T2			
TS7	T18	T11	T9	T19	T6	T15	T17	T14	T4	T1	
TS8	T8	T9	T4	T15	T6	T12	T19	T20	T17	T11	T2
TS9	T17	T6	T9	T14	T16	T19	T4	T7	T2	T3	
TS10	T3	T13	T6	T9	T4	T15	T1	T19	T18	T5	T10
TS11	T9	T4	T16	T8	T2	T6	T19	T18	T5	T10	
TS12	T4	T6	T19	T12	T15	T10	T8	T9			
TS13	T7	T9	T6	T18	T4	T3	T19	T16	T10	T11	
TS14	T20	T19	T9	T4	T6	T15	T5	T13	T12	T11	T3
TS15	T9	T4	T19	T16	T7	T10	T6	T12	T13	T8	
TS16	T16	T19	T4	T9	T11	T6	T14	T1	T8	T17	T2
TS17	T12	T15	T13	T4	T6	T8	T9	T1	T19	T17	
TS18	T2	T20	T6	T19	T16	T9	T17	T4	T14	T3	T1
TS19	T8	T6	T1	T15	T19	T7	T3	T9	T2	T4	
TS20	T6	T20	T5	T19	T2	T16	T4	T8	T9	T10	

The fourth step involves fitness calculation. The fitness value is determined by summation of total coverage of every test case in the test suite. The total coverage is then used to rank the test suites. The top order denotes greater coverage than the bottom one. The ranking is used to determine parents for Genetic Algorithm operators. (1) shows the calculation of fitness value [21]. The fitness values vary depending on the number of branches, statements and methods covered by the test suite. The more coverage, the higher the fitness value.

$$\text{Fitness} = \sum_{i=1}^{n} \frac{BT_i + BF_i + SC + MC}{(2*B) + S + M} \qquad (1)$$

Where

n represents the size of initial population,

i represents test cases ($1 < i < n$),

BT represents the number of branches evaluated to "true" once or more,

BF represents the number of branches evaluated to "false" once or more,

SC represents the number of covered statements,

MC represents the number of entered methods,

B represents the total number of branches,

S represents the total number of statements, and

M represents the total number of methods.

The example calculation below shows how the fitness value of test case T6 is computed. The resulted fitness value of T6 is 1.

$$\frac{1 + 3 + 13 + 1}{(2 * 6) + 5 + 1} = 1$$

The fifth step is applying the Genetic Algorithm. This step comprises four activities: (1) Test suite selection, (2) crossover, (3) mutation and (4) duplicate elimination. The performance of the Genetic Algorithm greatly depends on crossover and mutation. The test suites with high fitness value will be selected

to produce the offspring in the next generation. Generally, the worst half will be neglected. Thus, the best half is selected. After that, crossover is performed on both parents to produce a new test suite (offspring) that is better than the original test suites (parents). One point crossover is used. The crossover point is 2 as shown on figure 2. The bit strings after this point are exchanged. Figure 3 illustrates the two offsprings resulted from the crossover. Subsequently, the first offspring becomes a new test suite because it covers all independent paths. The selected offspring is shown on figure 4.

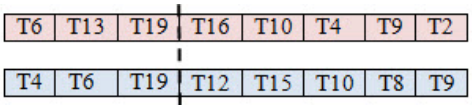

Fig. 2. Crossover is performed on two test suites

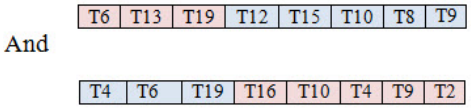

Fig. 3. The new offsprings resulted from crossover

Fig. 4. The selected offspring

Next, mutation is performed on a bit-by-bit basis to alter value of bits in a chromosome. The result is a new gene (test suite). Every bit in a chromosome of an offspring may change or mutate depending on mutation probability (*pm*). For each bit with in the chromosome, a random real number r is generated in the range [0, 1]. If r is less than *pm* then mutate the bit.

In this stage, two positions are randomly selected for swapping. The bits (test cases) can be swapped if they contain the identical path. If the test cases cover different paths, they are not allowed for swapping. This is to maintain the coverage. Figure 5 demonstrates this procedure. For example, T13 and T8 satisfy the condition. So they are swapped

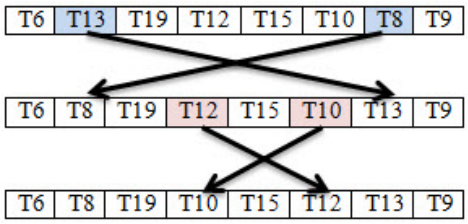

Fig. 5. The result of mutation

Lastly, duplicates are removed from the test suites to minimize the number of test cases. For example, T15 and T12 are removed. The minimized and prioritized test suite which covers all independent paths is shown on figure 6.

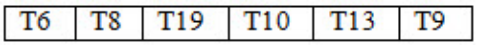

Fig. 6. The result of removing duplicates

B. *Evaluation*

The prioritized test suite resulting from the modified Genetic Algorithm is evaluated using the Average Percentage Code Coverage (APCC) metric which quantifies the degree at which a prioritized test suite covers the conditions.

The APCC is calculated as shown in (2).

$$APCC = 1 - \frac{\sum_{i=1}^{m} TC_m}{nm} + \frac{1}{2n} \quad (2)$$

Where T represents the test suite under evaluation,

n represents total number of test cases,

m represents the number of conditions under program (P), and

TC_i represents position of the first test case in test suite T to cover i^{th} condition.

TABLE III : APCC VALUES AND TIME SPENT FOR PRIORITIZTATION

Prioritization approach	APCC Value (%)	Execution Time (second)
No Order	94.50	0.050
Reverse Order	95.80	0.045
Random Order	99.83	0.020
Optimum Order	97.50	0.025
GA Order	**100.00**	**0.011**
BCO Order	94.70	0.030

Table III compares APCC values of each prioritization approach and time spent. In this paper, six prioritization approaches are studied. The first three approaches are considered as control groups where other conditions are identical to the other groups except the approach used. The prioritized test suite obtained from the Genetic Algorithm demonstrates complete code coverage (100%). Moreover, the time taken to execute the test suite generated by the modified genetic algorithm is significantly less than other prioritization approaches. Table IV presents orders of test cases resulted from all six prioritization approaches.

- No order : No technique is used. The test suite is unchanged and used on as is basis.
- Reverse order : The original order is reversed.
- Random order : The test cases are randomly ordered.
- Optimum order : The test cases are prioritized in the way that it maximizes the rate of fault detection.
- Genetic Algorithm : The modified genetic operators (crossover and mutation) are used to generate the prioritized test cases.
- Bee Colony Optimization : The test cases are prioritized to maximize the coverage and minimize execution time.

TABLE IV: THE RESULTS OF PRIORITIZATION OF DIFFERENT APPROACHES

No Order	Reverse Order	Random Order	Optimum Order	GA Order	BCO Order
T1	T20	T6	T9	T6	T17
T2	T19	T10	T7	T8	T19
T3	T18	T8	T4	T10	T18
T4	T17	T5	T2	T2	T2
T5	T16	T4	T18	T20	T20
T6	T15	T2	T1	T15	T15
T7	T14	T1	T13	T13	T13
T8	T13	T9	T3	T9	T9
T9	T12	T7	T5	T7	T7
T10	T11	T3	T8	T1	T1
T11	T10	T16	T6	T3	T3
T12	T9	T13	T10	T12	T12
T13	T8	T19	T19	T16	T16
T14	T7	T20	T11	T14	T14
T15	T6	T17	T15	T11	T11
T16	T5	T15	T12	T17	T6
T17	T4	T12	T16	T19	T10
T18	T3	T18	T14	T5	T5
T19	T2	T11	T17	T18	T8
T20	T1	T14	T20	T4	T4

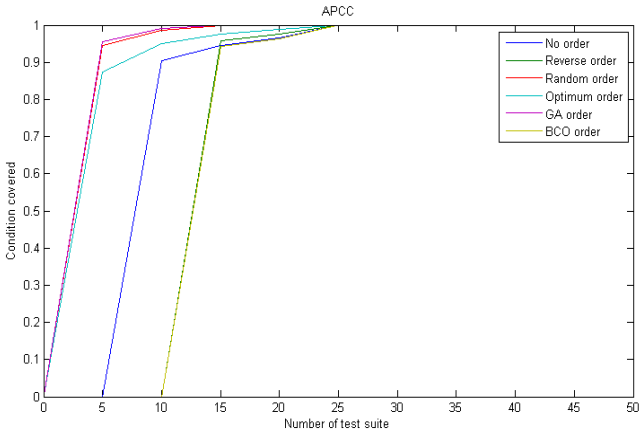

Fig. 7. The comparison of APCC Values for six prioritization approaches.

Figure 7 shows the relationship between the number of test cases executed and the corresponding APCC value. The figure indicates that 25 test cases are required to get APCC value to 100% when there is no order, or prioritization is done by Reverse Order and BCO approaches, while the modified Genetic Algorithm can get the APCC value to 100% from 15 test cases.

V. DISCUSSION

Genetic Algorithm clearly outperformed the other prioritization approaches both on APCC value and execution time. Although a BCO algorithm is found useful and efficient, it requires initial set up that is far more complex than the modified Genetic Algorithm. Table III reveals that the time spent on prioritization based on the BCO algorithm is almost three times the modified Genetic Algorithm. Unsurprisingly, the unordered test suite comes last on the execution time. It takes almost five times longer than the modified Genetic Algorithm.

Prior to the experiment, the performance of the test suite with Reverse order and Random order were thought to be similar to No order approach. While it holds true for the Reverse order, it is not for the Random order in this example. The reason behind this is that the top order of the Random ordered test cases appears to be similar to the order generated via the modified Genetic Algorithm.

VI. CONCLUSIONS AND FUTURE WORK

Test case prioritization is continuously proven to be beneficial to Regression testing. In this paper, the Genetic Algorithm is modified to prioritize test cases to yield the maximum code coverage. Six approaches to prioritizing test cases are empirically studied and their performances are compared. The performance of the modified Genetic Algorithm both on the coverage and time criteria shows a promising outcome.

The modified Genetic Algorithm takes advantage of simplicity and ability to vary the population to provide a variety of test cases for selection and prioritization processes. Inputs to Genetic Algorithm are also important. A noble output is often a result of a decent input. Similarly, the test cases generated as input to the modified Genetic Algorithm are randomized based on one condition that each of them covers all independent paths of the program to assure the highest coverage.

The Bee Colony Algorithm (BCO) is a sound technique for test case prioritization. However, the algorithm is more complicated than a Genetic Algorithm. It takes longer time to obtain the optimum test cases. Therefore, it is not a good candidate for test case prioritization, especially when the program gets bigger.

The results obtained from this study advise that the further studies should be done. Firstly, bigger programs or programs with more complex decision graphs should be studied and compared with the results in this paper. Additional prioritization approaches should be included in the future studies to find out if it will perform better than the modified Genetic Algorithm, or the complex algorithm such as BCO will perform better in different types of programs.

REFERENCES

[1] S. Yoo, M. Harman. Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22.2 (2012): pp. 67-120.

[2] J. Hwang, et al. Selection of regression system tests for security policy evolution." Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, 2012.

[3] A. Kaur, S. Goyal. A genetic algorithm for regression test case prioritization using code coverage. International journal on computer science and engineering 3.5 (2011) pp. 1839-1847.

[4] S.R. Khan, A. Nadeem, TestFilter: A Statement-Coverage Based Test Case Reduction Technique. IEEE Conference on multitopic, (INMC' 06), pp.275-280, December 2009.

[5] M.A. Nada and AL-Salami, Evolutionary Algorithm Defintion, American Journal of Engineering and Applied Sciences, Vol.2, pp. 789-795, 2009

[6] E. Engstrom and P Runeson, A Qualitative Survey of Regression Testing Practices, Springer-Verlag Berlin heidelbreg, LNCS 6156, pp. 3-16, 2010

[7] A. Askarunisa, L. Shanmugapriya, and N. Ramaraj, "Cost and Coverage Metrics for Measuring the Effectiveness of Test Case Prioritization Techniques, INFOCOMP Journal of Computer Science, pp. 1-10, 2009

[8] M. Athar, I. Ahmad, Maximize the Code Coverage for Test Suit by Genetic Algorithm, International Journal of Computer Science and Information Technologies, Vol.5, pp. 431-435, 2014

[9] P.R. Srivastava and T-h. Kim, Application of Genetic Algorithm in Software Testing, Internationl Journal of Software Engineering and Its Applications, Vol.3, October 2009

[10] P.B. Nirpal and K.V. Kale, Using Genetic Algorithm for Automated Efficient Software Test Case Generation for Path Testing, Internationl Journal of Advanced Networking and Applications , Volume: 02, pp. 911-915 , 2011

[11] G. Fraser and A. Arcuri, Whole test suite generation, Software Engineering, IEEE Transactions on 39.2, pp. 276-291, 2013

[12] A. A. Ahmed, Dr. M. Shaheen, and De. E. Kosba, Software Test suite prioritization using multi-criteria fitness function, Computer Theory and Applications (ICCTA), 2012 22nd International Conference on. IEEE, pp. 160-166, October, 13-15 2012

[13] S. Elbaum, A. G. Malishevsky, G. Rothermel. Prioritizing test cases for regression testing. Vol. 25. No. 5. ACM, 2000.

[14] Z. Li, M. Harman, R. M. Hierons. Search algorithms for regression test case prioritization. Software Engineering, IEEE Transactions on 33.4 (2007): 225-237.

[15] K. Aggrawal, Y. Singh, A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. ACM SIGSOFT Software Engineering Notes 29.5 (2004): pp. 1-4.

[16] G. G. S. Indraprastha. A bee colony optimization algorithm for fault coverage based regression test suite prioritization. Organization 29 (2011).

[17] L. Davis, editor. Handbook of Genetic Algorithms. Van Nostrand Reinhold, New York, 1991.

[18] D. E. Goldberg, editor, "Genetic Algorithms in Search, Optimization, and Machine Learning", Van Nostrand Reinhold, New York, 1991.

[19] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey", IEEE Computer, 27(6), pp.17–26, June 1994.

[20] J. L. R. Filho, P. C. Treleaven, and C. Alippi, "Genetic-algorithm programming environments", IEEE Computer, 27(6), pp.28–43, June 1994.

[21] R. Jameson.(2014, April 11). About Code Coverage[Online]. Available: https://confluence.atlassian.com/display/CLOVER/About+Code+Coverage