Test Case Prioritization based on Historical Failure Patterns using ABC and GA

Pushkar Padmnav
Aricent
Gurgaon, India
pushkarpadmnav63@gmail.com

Gaurav Pahwa
Aricent
Gurgaon, India
gaurav.pahwa@aricent.com

Dinesh Singh
Aricent
Gurgaon, India
dinesh.singh@aricent.com

Saurabh Bansal
Aricent
Gurgaon, India
saurabh.bansal@aricent.com

***Abstract*— Regression testing is defined as type of software testing carried out to ensure that no new bugs are introduced due to modifications to existing code or addition of new features. Prioritization techniques order the set of test cases for improved & effective testing. Test Case Prioritization (TCP) in corrective regression testing is an indispensable arsenal to help discover faults faster during the initial phase of testing. Many techniques have been proposed for Test Case Prioritization (TCP) based on requirement correlation, test coverage, information retrieval which are dependent on data which is not easily available. Our approach considers the historical execution of the regression cycles through use of Artificial Bee Colony Optimization (Swarm Intelligence) & Genetic Algorithm for fault detection with improved results.**

Keywords*— *Test Case Prioritization, Regression Testing, APFD, Artificial Bee Colony Optimization, Genetic algorithm, Corrective Regression Testing, Swarm Intelligence

I. INTRODUCTION

Testing is an integral part of the software development lifecycle and ensures product's reliability, compatibility, efficiency and robustness. Testing is an activity to check whether the actual results match the expected results and software is defect free [1]. There are close to ~100 testing strategies [2] which exist today, the most common ones used by various companies are unit testing, integration testing, regression testing, smoke testing, alpha testing, beta testing, stress and performance testing. It has been observed that at least 40-50% of the effort is expended in testing a software.

A specification (requirements) or structural (code) change can introduce bugs which are detected during the regression testing. Although, regression is an important exercise for testing a product for its robustness, the time it takes for testing a product can span from days to weeks for a complete set of execution. The faults, if revealed late in the cycle delays bug fixes and project releases. Prioritizing test cases helps in improving the performance of the testing cycle through detection of faults earlier in the cycle, thereby permitting more time for troubleshooting and debugging of issues.

Multiple approaches have been researched for test case prioritization using traditional code coverage based techniques which have not proved to be very effective [3]. Techniques based on requirement correlation and weights procedure [4] are better than code coverage based prioritization. However, the unavailability of the mapping of the test cases to the requirements (requirements traceability) and effort to pull such information poses a serious challenge to already struggling project deliveries and timelines. To address above issue, we have proposed a swarm & GA based artificial intelligence technique which utilizes the historical regression testing data to generate the prioritized set of test cases based on the number of faults it uncovers and its execution time with better results.

Section II describes the background on testing techniques. Then we briefly describe the evolutionary and swarm algorithms in Section III. Section IV details out the experimental analysis. In Section V, we discuss the experiment results. Lastly, we present our future work in Section VI and conclusion in Section VII.

II. BACKGROUND

A. Regression Testing

Regression testing is a process which is applied after a program is modified. It tests the modified program to build confidence that the changed program will perform as per the specification (possibly modified). Regression forms an integral part of the testing strategy during the maintenance phase where the software system may be corrected, adapted to new environment, or enhanced to improve its performance [7]. As the regression suites grow with each new-found defect, test automation is frequently involved which often poses a challenge due to time constraints.

B. Types Of Regression Testing

Corrective regression testing and progressive regression testing are forms of regression testing [7]. Whenever a new requirement is incorporated into a system, the specification is modified. Progressive testing tests the program for its correctness against the modified specification. The cycle is invoked at regular intervals and fewer test cases can be reused during the testing. However, conducting this testing helps in ensuring that, there are no features that have been compromised in the new and updated version that exists in the previous version. In Corrective Regression testing, the specification doesn't change. Only few areas of code may be modified for any fixes. According to [21], fixes involve

978-1-5386-5933-5/19/$31.00 ©2019 IEEE

software failures, performance failures, and implementation failures to keep the system working properly. The set of test cases from the previous test plan will be valid but might not be able to test the previously targeted program constructs during the corrective regression testing.

Progressive testing is done after enhancements related to specification change are carried out, while corrective regression testing is done during development cycle and after corrective maintenance. It is imperative to understand that both these types of regression testing involve execution of two set of test cases viz. specification based test cases and structural test cases. The approach in the paper is intended for corrective regression testing.

III. TEST CASE PRIOTIZATION PROPOSED APPROACH

A. Motivation

Test case prioritization has been explored across many research papers through use of multiple techniques and empirical studies have supported the methodologies used.

Charitha et al. [4] proposed prioritizing the test cases based on requirements and risks. The technique uses risk level of potential defect type to identify the risky requirements and then prioritizing the test cases by establishing the relation between test cases and the requirements. E.g. A security threat defect will correspond to nonfunctional requirements which will have multiple test cases associated and depending on the risk severity of the defect, the prioritization will be done.

Arafeen et al. [9] suggested the use of requirements based clustering using similarity measure and selecting the test cases from each cluster following specific methods.

Jung-Hyun Kwon et al. [10] tried to address the limitations of the traditional code coverage techniques for infrequently tested code by use of Information Retrieval Concepts. Using coverage score and similarity score features calculated by TF (Term Frequency) and IDF (Inverse Document Frequency), the linear regression model is trained and evaluated to assign weights to the test cases based on both features.

All the above approaches rely either on source code information, or factors related to requirement such as implementation complexity and customer priority. However, our approach is based on the historical regression data and execution time of the test cases which is captured through various test management tools.

B. Understanding Swarm Intelligence

Swarm Intelligence is a collective behavior exhibited by group of organisms or artificial systems with ability to manage complex systems of interacting individuals through minimal communication to produce a global emergent behavior [12]. These systems are self-organizing and achieve their objectives through the interactions of the entire group. Bee swarming, ant colonies, fish schooling are some natural swarm intelligence systems. These groups are smarter when they are thinking together, than they would be individuals on their own. Swarm requires no leader. There is no single ant or bee influencing the decisions to exploit a food source or create trails for movement.

Modern science has been researching on the swarm capabilities to build systems which are autonomous in nature and are more superior and capable than a single intelligent unit. These properties of collaboration, cooperation and learning from biological systems can be employed to solve several optimization problems in various domains.

C. ABC (Artificial Bee Colony Optimization) and GA (Genetic Algorithm)

In our experiment, we have used the combination of Artificial Bee Colony Algorithm (ABC) which is a swarm inspired meta-heuristic algorithm and evolutionary method (Genetic algorithm) [11] for prioritizing the test cases. Metaheuristic is a high-level framework to develop heuristic optimization algorithms that may provide a sufficient good solution to an optimization problem especially with incomplete information. Metaheuristics can often find good solutions with less computational effort than optimization algorithm, iterative methods or simple heuristics. Evolutionary computation, genetic algorithms and particle swarm optimization [14] are population based metaheuristics. Since its introduction by Karaboga in 2005 [15] for solving numerical optimization problems, Artificial Bee Colony Algorithm (ABC) has stirred a lot of interest among researchers. Two vital notions of self-organization and division of labor are essential to obtain swarm intelligence behavior for system to self-organize and adapt to given environment as put across by Karaboga.

- Self-organization is a set of rules for interactions between the components of the system. These are essentially positive feedback, negative feedback, fluctuations and multiple interactions.
- Division of labor is segregation of different set of tasks to be performed by specialized individuals which helps swarm to respond to changed environments in search space.

In Artificial Bee Colony Algorithm, there are three components: food sources, employed bees and unemployed bees. Unemployed bees are grouped into either onlooker or scout bees.

The food sources are chosen according to the distance from the nest, nectar amount which can also be termed as profitability or fitness value of the source. Employed bees or foragers are associated with a food sources where they are employed. These bees share the fitness information with the other group of bees by performing a waggle dance. The other group of bees namely scouts are on a search for new food sources around the nest and onlooker bees wait in the nest for establishment of the most profitable food source from the information shared by the employed bees.

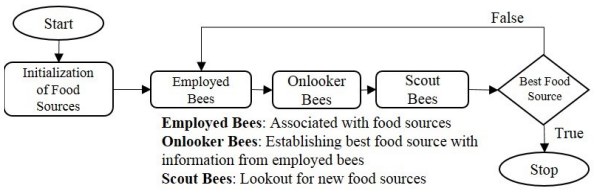


Fig. 1: ABC process

Genetic Algorithm have been used for problem solving since its inception, but not on a larger scale due to the time complexity it carries. However, with the advent of fast computers, the computational power is not a challenge anymore [11]. GA used the analogy of the natural evolution where the strongest of all are most likely to survive.

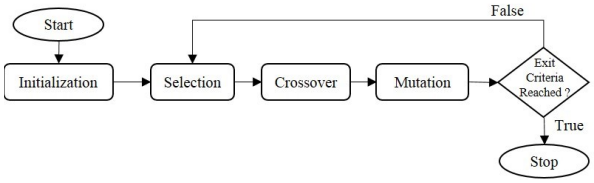


Fig. 2 : GA Process

The process of GA begins with initializing a polulation, then selecting the parents to create a new set of offspring through cross over. The mutation is applied to each of the offspring after crossover. The desired output is to yield a fitter set of offspring or a better solution to a problem. The cycle continues till the exit criteria is met i.e. solution reached for a problem. The usage of ABC and GA has been eloborated in the experimental evaluation section

IV. Experimental Evalutaion

Having laid down the fundamentals of the Swarm intelligence, ABC and GA algorithms, we will now detail out the Test case prioritization using ABC and GA with the historical execution data from the dataset.

A. Metrics and Evaluation Criteria

APFD is a standard metrics which can be used to evaluate the performance of the prioritization technique. APFD stands for Average Percentage Faults detected [10] which basically means how quickly the faults are detected during a test execution cycle.

$$\text{APFD} = 1 - \frac{TF_1 + TF_2 + \ldots . TF_m}{nm} + \frac{1}{2n} \quad (1)$$

where
TF_i → the position of the first test in T that exposes the fault i
m → number of the faults in the program
n → number of the test cases
The APFD metrics value ranges from 0 – 1. The value closer to 1 indicates that faults are detected early in the execution of the prioritized cycle.

B. Experimental setup

Open industrial data set from ABB Robotics Norway [18] has been used for Test Case Prioritization. Out of the three data sets available - Paint control, IOF/ROL and GSDTSR (google shared dataset for test suite results), IOF/ROL has been selected as an input to the experiment.

a) Dataset

The dataset is available on bitbucket[1] with following features.

TABLE I: Dataset Features

Column	Description
Cycle	Cycle number to which the test execution belongs
Id	Unique Id for each execution
Duration	Execution time of the test run
Name	Test case name
Verdict	Test verdict (Failed: True, Passed: False)
Last results	List of previous results
Last run	Last execution of test case in timestamp format

The following features are filtered to select only relevant to the TCP algorithm.

TABLE II: Filtered Features

Column	Description
Test Execution record	Unique incrementing value
Test case id	Test case ID / name
Time Taken	Execution time in milliseconds
Result	Test Verdict (Failed: True, Passed: False)

There are total 2086 unique test cases and 32260 execution records for 320 execution cycles in the dataset selected.

b) Data Selection

Finding an open regression test dataset has been a challenge for the experiment. The dataset[1] cycles were not exclusively regression and had mixed execution including specific test case selection for feature testing. The maximum number of test cases executions in any of the cycles was found to be 1064. 10 cycles were identified in order of decreasing order of the number of test records execution. The identified cycles are as follows.

TABLE III: Training & Validation data selection

S.No.	Cycle	Number of executions	Unique test cases	Training cycle	Validation cycle
1	178	1064	1058	177	178
2	180	965	949	179	180
3	245	935	935	244	245
4	163	806	774	162	163
5	249	792	175	248	249
6	263	788	558	262	263
7	103	757	757	102	103
8	270	748	667	269	270
9	275	735	680	274	275
10	67	690	690	66	67

[1] Data set extracted from https://bitbucket.org/helges/atcs-data

c) Training Procedure

In this section, we use the test cases data from the Table III in the order as specified. For each of the historically executed cycles, the data has been trained and then validated on the last cycle for measuring its efficiency in detecting the faults through the APFD metrics.

The bees start foraging for each of the test cycle to prioritize the test cases. The process starts by initial foraging with half bees as number of test cases while adding a test case to each of the bee. The Table IV matrix gives an overview of a sample execution cycle of test cases. The table captures six days of execution history which can span for more than a year. The data is from 01-Sept-2018 till 06-Sept-18, where the columns represent the dates for execution of test cases T1 to T4.

Table IV: Test Case Execution Matrix over 6 days

Test Case	09/01	09/02	09/03	09/04	09/05	09/06
T1	1	0	0	0	0	0
T2	0	0	1	0	1	0
T3	0	1	0	0	0	1
T4	1	0	0	0	1	1

TABLE V: Test Case Execution Time

Test Case	Execution Time
T1	6
T2	2
T3	4
T4	4

The 1's are the test failures and 0's are the pass values. In the first round the bees return to hive but no crossover is done as each bee has only a single test case. However, the exit criteria are applied to verify if any bee has discovered all faults.

If the fitness value by adding a new test case to a bee increases i.e. more number of 1's through OR operation, the test case is added else the cycle continues till the remaining list is exhausted. On successful addition of a new test case, the bees return to hive for crossover. The new set of offspring are only added to the set of foraging bees, if the following criteria of crossover are met.

- None of the existing bees should have the same state as that of the newly created offspring i.e. same number of 1's.
- Execution time of the new bees should be less than the maximum execution time of the any other foraging bee.

If any of the bees discovers all failures or has exhausted all the test cases, the exit criteria are met and the process ends. At this juncture, the bee will perform waggle dance to announce the best food source i.e. the best solution for Test Case Prioritization. The number of test cases for that bee are ranked according to the minimum execution time and are subtracted from the total test cases. The process is then again repeated for the remaining test cases till all the test cases are ranked or prioritized.
More weightage is allotted to a test case with minimum execution time, if two test cases have same fault discovery profile. Fault discovery profile or fitness value refers to the goodness of the test case for solving a problem. The more the number of 1's for a foraging bee, better the fitness value.

For the best results, the complete cycle is repeated five times and weighted average of the ranks is considered as the final ranks for the prioritized test cases. Finally, the ranked test cases are grouped into five prioritized buckets P1 through P5. P1 containing the highest prioritized list of test cases. The Fig. 3 gives an overview of the process.

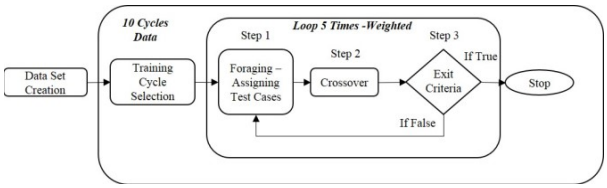


Fig. 3: Training process with ABC and GA

TABLE VI: Initial Assignment

Bee	Test Case	Test Case Fault Detecting Capacity	Fitness Value	Time
B1	T2	001010	2	2
B2	T3	010001	2	4

Step 1:

TABLE VII: Test Case Assignment

Bee	Test Case	Test Case Fault Detecting Capacity	Fitness Value	Time
B1	T2, T1	101010	3	8
B2	T3, T4	110011	4	8

Step 2:

B3: T2 | T4

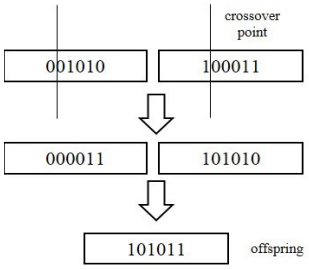


TABLE VIII: Crossover

Bee	Test Case	Test Case Fault Detecting Capacity	Fitness Value	Time
B1	T2, T1	101010	3	8
B2	T3, T4	110011	4	8
B3	T2, T4	101011	4	6
B4	T3, T1	110001	3	10

Bee B4 will not be added to the list as its execution time is not meeting the crossover criteria. The Bees will again forage for addition of new set of test cases

TABLE IX: Foraging

Bee	Test Case	Test Case Failure Detecting Capacity	Fitness Value	Time
B1	T2, T1, T3	111011	5	12
B2	T3, T4, T1	110011	4	14
B3	T2, T4, T3	111011	5	10

In this cycle, the Bee B3 will not be assigned the test case T1 as it doesn't add to its fitness value. Now adding T3 to B2 will have the exit criteria met. The Bee B3 has minimum execution time and shares same fitness with B1, hence B3 will be chosen for first set of prioritized test case.

The prioritized test cases will be T2, T3, T4 & T1 in that order.

V. Experiment Results

a) Validation

The results of the prioritization for each of the cycle in Table III are divided into buckets from P1 to P5. P1 contains the highest rank test cases. Each validation cycle is then verified for fault detection against model generated for each cycle. As seen from the results, the failures for each of the validation cycle are detected in top most prioritized buckets.

TABLE X: First Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	92	74	50.68	196
P2	214	51	85.61	97
P3	226	11	93.15	124
P4	78	7	97.94	275
P5	302	3	100	56

Table X contains the validation result for the first cycle. On examining the results, it shows that around 85% of the failures are detected during the execution of the P1 & P2 bucket test cases. The APFD value for the above set of prioritizations improved from default value of 0.54 to 0.78. Here, the total number of test cases trained were 1806. Out of the 1806 test cases, 1064 were executed in the 178th cycle. The sum of Passed, Failed and not Executed test cases is 1806. The sum of Passed and Failed is 1058 as there were few multiple executions of the test case in the same validation cycle.

Similarly, the tables through XI – XIX presents the result of the validation for all the other nine cycles with desirable outcome.

However, there are three validation cycles where the failures are detected late in the cycle in P5 bucket. These are the cycles where it has been observed that the test cases falling in the P5 bucket are the ones which never failed previously and were low prioritized but failed in the validation cycle. A mixed solution approach will be discussed for handling this edge case in the Future Work section. The other set of cycles present excellent results and a drastic improvement in APFD.

TABLE XI: Second Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	52	67	17.58	243
P2	155	84	39.63	122
P3	157	64	56.43	140
P4	54	12	59.58	295
P5	150	154	100	57

TABLE XII: Third Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	100	20	50	254
P2	188	16	90	169
P3	204	2	95	168
P4	225	1	97.5	147
P5	173	1	100	200

TABLE XIII: Fourth Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	62	33	38.37	260
P2	148	34	77.9	173
P3	177	18	98.83	160
P4	71	0	98.83	284
P5	230	1	100	124

TABLE XIV: Fifth Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	3	24	16.32	348
P2	4	19	29.25	351
P3	12	13	38.09	350
P4	4	13	46.93	357
P5	0	78	100	297

TABLE XV: Sixth Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	30	56	33.73	294
P2	60	20	45.78	300
P3	99	12	53.02	268
P4	107	54	85.54	219
P5	89	24	100	267

TABLE XVI: Seventh Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	91	37	84.09	180
P2	143	3	90.90	161
P3	168	0	90.90	139
P4	116	0	90.90	191
P5	194	4	100	110

TABLE XVII: Eight Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	78	38	35.84	267
P2	155	24	58.49	204
P3	130	38	94.33	214
P4	139	6	100	238
P5	59	0	100	324

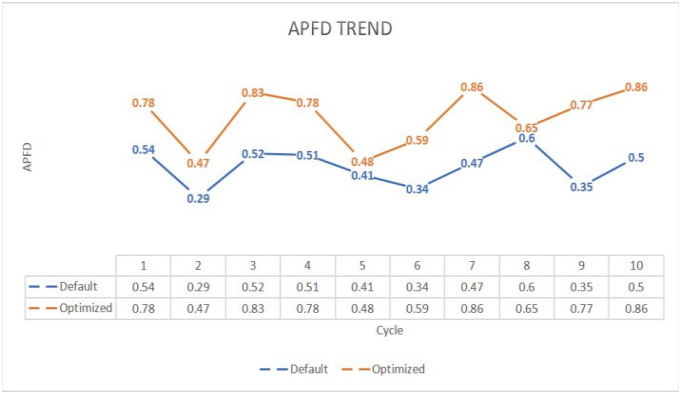
TABLE XVIII: Ninth Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	35	81	47.64	269
P2	82	88	99.41	215
P3	99	1	100	285
P4	71	0	100	314
P5	101	0	100	284

TABLE XIX: Tenth Cycle Validation Results

Priority	Passed Test Cases	Failed Test Case	Cumulative Detection %	Not Executed
P1	70	41	73.21	108
P2	166	11	92.85	41
P3	159	2	96.42	57
P4	132	2	100	84
P5	107	0	100	111

The graph illustrates the APFD trend for the ten cycles of validation against the trained prioritized model. The average of the APFD stands at 0.70 for all cycles and 0.79 for cycles other than three identified in results section where failures were detected in last cycle.



	1	2	3	4	5	6	7	8	9	10
Default	0.54	0.29	0.52	0.51	0.41	0.34	0.47	0.6	0.35	0.5
Optimized	0.78	0.47	0.83	0.78	0.48	0.59	0.86	0.65	0.77	0.86

Fig. 4 : APFD Trend

VI. Future Work

Proposed TCP technique ranks the test cases in order of their effectiveness to uncover faults purely based on its execution history. It does not have knowledge of new features introduced in the system that can also lead to failure of regression test cases that do not have prior history of failure. Future work is planned to undertake test case selection to determine the subset of regression test cases that are potentially affected by a given modification to the system.

Additionally, it is planned to further optimize the TCP by considering the severity of the bug associated with the failed test cases. This can be achieved by incorporating bug severity weight during the training cycle. Test case failures associated with higher severity bugs will carry a higher weightage.

VII. Conclusion

In this paper, we proposed a novel approach for prioritization of the test cases using the historical failure patterns for the correction regression cycle testing. The results clearly indicate that there is significant improvement in the fault detection capability at the start of the regression testing itself. The training data used by our model is mostly maintained by the organization through various set of test management tools. Our technique relied on the past execution cycles and has no dependency on the requirements, feature risks, code coverage analysis, thereby reducing the efforts for a software company to invest time in collection & preparation of the data for the training.

Acknowledgment

This research was supported by Aricent Technology Innovation.

References

[1] G. Saini, K. Rai, “An Analysis on Objectives, Importance and Types of Software Testing”

[2] https://www.softwaretestinghelp.com/types-of-software-testing/

[3] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness”

[4] C. Hettiarachchi, H. Do, B. Choi , “Effective Regression Testing Using Requirements and Risks ”

[5] J. Hartmann, D.J. Robson, “Approaches to regression testing”

[6] R.R. Bate, G.T. Ligler, “An approach to software testing: methodology and tools”

[7] H.K.N. Leung, L. White , “Insights into regression testing (software testing)”

[8] J. Gaur, A. Goyal, T. Choudhury, S. Sabitha, “A walk through of software testing techniques”

[9] Md. J. Arafeen, H. Do , “Test Case Prioritization Using Requirements-Based Clustering”

[10] J.-H. Kwon, I.-Y. Ko, G. Rothermel, M. Staats, “Test Case Prioritization Based on Information Retrieval Concepts”

[11] K.F. Man, K.S. Tang, S. Kwong , “Genetic algorithms: concepts and applications [in engineering design]”

[12] Y.-F. Zhu, X.-M. Tang , “Overview of Swarm intelligence”

[13] Y. Yi, R. He , “A Novel Artificial Bee Colony Algorithm”

[14] I. Koohi, V. Z. Groza , “Optimizing Particle Swarm Optimization Algorithm”

[15] D. KARABOGA, “AN IDEA BASED ON HONEY BEE SWARM FOR NUMERICAL OPTIMIZATION”

[16] Md. I. Kayes, “Test case prioritization for regression testing based on fault dependency”

[17] R. Gulati, P. Vats , “A literature review of Bee Colony optimization algorithms”

[18] H. Spieker, A. Gotlieb, D. Marijan , M. Mossige , “Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration”

[19] B. Suri, I. Mangal, V. Srivastava, “Regression Test Suite Reduction using a Hybrid Technique Based on BCO And Genetic Algorithm”

[20] https://www.testbytes.net/blog/types-of-regression-testing/

[21] B. P. Lientz and E. B. Swanson, “Software Maintenance Management”, Addison-Wesley, 1980

[22] https://en.wikipedia.org/wiki/Metaheuristic