International Journal of Soft Computing 11 (4): 247-254, 2016
ISSN: 1816-9503
© Medwell Journals, 2016

Regression Test Cases Prioritization for Object-Oriented Programs Using Genetic Algorithm with Reduced Value of Fault Severity

Samaila Musa, Abu Bakar Md Sultan, Abdul Azim Bin Abd-Ghani and Salmi Baharom
Faculty of Computer Science and Information Technology, Universiti Putra Malaysia,
43400 Serdang, Malaysia

Abstract: One of the most important activities in software maintenance is Regression testing. The re-execution of all test cases during the regression testing is costly. Several of the researchers address the issue of test case prioritization using genetic algorithm, but most of the researcher do not select modification-revealing before prioritization and used the same fault severity. This stusy presents regression test case prioritization of selected test cases for object-oriented software using genetic algorithm with reduced fault severities when a fault is executed by the preceding test cases. The experimental evaluation of our proposed approach was done using nine programs. The researcher measure the performances of their prioritization approaches using Average Percentage of rate of Faults Detection (APFD) metric. It was observed from the results that the approach increases the effectiveness of regression testing in term of rate of fault detection. It was concluded that prioritization of modification-revealing test cases based reduced fault severity to compute fitness value to a smaller value provides considerably better results.

Key words: Regression testing, genetic algorithm, test case prioritization, APFD, severity

INTRODUCTION

Software testing is one of the software engineering activities that is perform even after the software delivery. Regression testing is a software activity performs due to some changes in the software in term of improvement due to user's requirements or debugging to make sure that existing functionalities and the initial requirements of the design are not affected. But, it is not feasible to retest-all due to cost and time consumption.

Regression testing is an activity that tests the modified program to ensure that modified parts behave as intended and the modification have not introduced sudden faults. Regression test selection technique helps in selecting a subset of test cases from the test suite. After the selection, when the selected test cases are large in number, there is need to order them based on some criteria to reduce the execution time of regression testing, i.e. prioritization of the test cases. Rothermel *et al.* (2001) describe regression test case prioritization as a technique that allows the testers to order their test cases based on certain criterion so that those with highest priority are executed earlier in regression testing activity than lower priority test cases and it can be used in conjunction with test case selection when test case discarding is acceptable, also assert that it can increase the utilization of testing time more beneficial than non-prioritize when the regression testing activities are unexpectedly terminated.

The regression test case selection and prioritization can be made using different approaches, either using heuristic approach or evolutionary approach. But using evolutionary approach, example genetic gorithm gives better results in making sure that all affected parts are tested. The effectiveness of genetic algorithm based approaches was proved by a lot of researcher. Elbaum *et al.* (2002) proposed an approach that focuses on scheduling the test cases so as to improve the performance of regression testing. The author explained the four methodologies of regression testing: retest all, regression test selection, test suite reduction and test case prioritization and it was proved that prioritized test cases show better results than the non-prioritized test cases between prioritized and non-prioritized one using Average Percentage of Fault Detected (APFD).

Huang *et al.* (2010) proposed an approach that considers the cost-cognizant metric that used varying test case costs and fault severities for test case prioritization technique based on the use of historical records and a genetic algorithm. The authors run a controlled experiment to evaluate the effectiveness of the proposed technique. The results indicate that the proposed technique

Corresponding Author: Samaila Musa, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, 43400, Malaysia

frequently yields a higher Average Percentage of Faults Detected per Cost (APFDc) and also show that it is also useful in terms of APFDc when all test case costs and fault severities are uniform. Malhotra and Bharadwaj in 2012 proposed a prioritization approach based on the execution history of test cases using GA. The fitter test cases based on the number of statements covered were given higher priorities. But the approach is for procedural and smaller program.

You and Lu (2012) proposed an approach based on genetic algorithm for the time aware regression testing reduction problem. The maximization of the execution time of the remaining test cases after the removal of the redundant test cases was the main objective of this problem. The authors used eight problems as example to evaluate the genetic algorithm to examine the efficiency. A multiple control flow based coverage criteria that improved test suite prioritization technique of structural program testing by increasing the test code coverage was proposed by Ahmed *et al.* (2012) that maximize the number of coverage items. The authors considered test suite as chromosome, test cases as genes and assigned weights to the test cases to compute fitness. They evaluate performance of the approach using Average Percentage of Fault Detected (APFD) and the approach shows better results.

A requirement based system level test case prioritization was proposed by Kumar and Chauhan (2013) and Raju and Uma (2012) in order to reveal severer faults at an earlier stage. This is based on factor oriented regression testing using GA. A novel approach based on rule based fuzzy classification was proposed by Xu *et al.* (2013) that selects set of high effective test cases from a very large pool. The authors also developed a test plan for large real world software systems using the rule based fuzzy classification. The rules are generated for only if-then. A prioritization approach using genetic algorithm to order test cases was proposed based on fault coverage and execution time by Singh and Kaur (2014). The approach was not implemented. In two similar previous approaches (Musa *et al.*, 2014a, b), the authors presented evolutionary approaches that used genetic algorithm, but the changes were made manually which may result in making bias selection, and the initial population in GA are only two parents which reduce the possibility of forming best ordering. Also the fitness value was computed using the same fault severity history.

Purohit and Sherry (2014) proposed an approach that used genetic algorithm to prioritize test suites. The approach prioritize test suites based on their fitness value, and the fitness values are computed based on the number faults detected. But the approach does not consider dependencies between the changed and other statements in order to find statements that are dependent on the changed statements. Also the changes might not have affected all the test cases, there is need to select affected test cases and then prioritize them. Their approach is for procedure programs, therefore it cannot be apply directly apply for object-oriented programs. They also used the same severity even if a fault was executed by the previous test case.

In this study we present an evolutionary prioritization approach that prioritizes the selected test cases by using genetic algorithm. The fitness value is computed using reduced severity of fault of already executed statement by the preceding test cases.

MATERIALS AND METHODS

This study describes genetic Algorithm, the evaluation matric (APFD), regression testing and experimental design.

Genetic algorithm: Many real life problems have been solved using evolutionary algorithms and GA is one such evolutionary algorithm. Genetic Algorithm has emerged as optimization technique and search method. Problems being solved by GA are represented by a population of chromosomes as the solution to the problems in form of string of binary digits, integer, real or characters and each string that makes up a chromosome is called a gene.

Average percentage of rate of faults detection (APFD) metric: To evaluate the performance of regression test case prioritization, many researchers Musa *et al.* (2014a, b), Kumar and Chauhan (2013), Xu *et al.* (2013), Raperia and Srivastava (2013), Panigrahi and Mall and Gupta and Yadav (2013) used APFD metric to evaluate the prioritization techniques. A sample of test cases and the faults detected by each test case are presented in Table 1.

The APFD metric widely used in gauging the performance of program P and test suite T is given as:

$$APFD = 1-(Tf_1+Tf_2+\ldots+Tf_m)/nm+1/2n$$

Table 1: Faults detected by the test cases

Test casesFaults	T1	T2	T3	T4	T5	T6	T7	T8
F1		x						
F 2		X				x		
F3	x		x	x				
F 4				x	x	x		
F5					x	x		

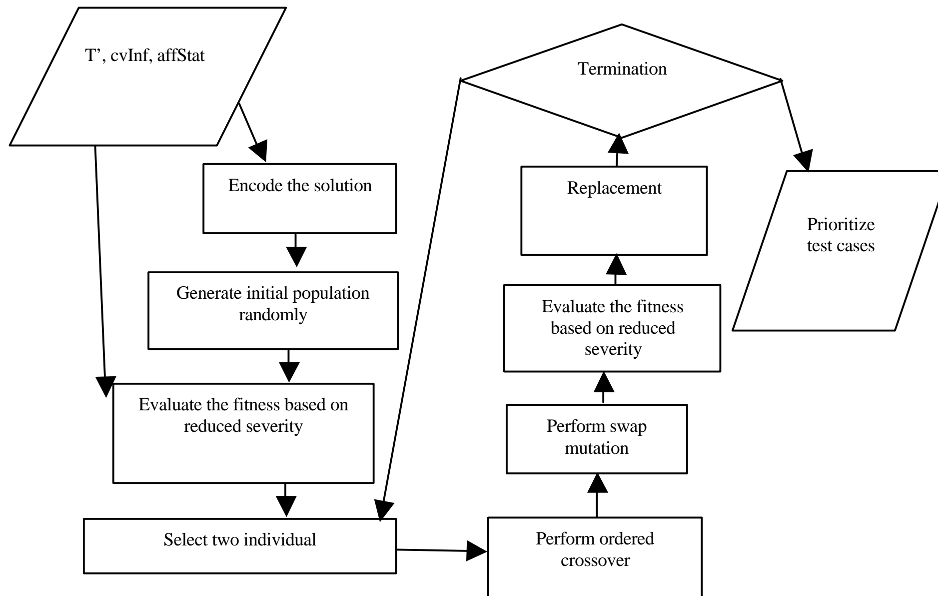

Fig. 1: Our proposed prioritization framework

Where:

m = The number of faults

n = The number of test cases

$Tf_1+Tf_2+...+Tf_m$ = The position of the first test in T that exposes the faults 1, 2… m

Consider the ordering:

- T6, T5, T1, T2, T3, T4, T7, T8
- T6, T2, T1, T5, T3, T4, T7, T8

Solution

T6, T5, T1, T2, T3, T4, T7, T8:

APFD = 1-(1*4+1*3+1*1+1*1+1*1) / 8*5+1 / 2*8

= 1–10/40+1/16

= 1–0.25 + 0.0625

= 0.8125 = 81.25%

T6, T2, T1, T5, T3, T4, T7, T8:

APFD =1-(1*2+1*2+1*1+1*1+1*1) / 8*5+1 / 2*8

= 1–7/40+1/16

= 1–0.175 + 0.0625

= 0.8875 = 88.75%

The higher the value of APFD, the better the value, the faster the rate at which faults are detected and less the cost of regression testing of modified programs.

Regression testing: This study presents proposed approach for the prioritization of selected test cases T'. Our proposed approach is based on GA with reduced fitness value of already executed fault by the preceding test case. The proposed approach is shown in Fig. 1.

Test case prioritization: This section presents our proposed approach. The approach uses GA to optimize the selected test cases. The selected test cases T` are prioritized using genetic algorithm (Algorithm I), denoted as "ModGAPriorTCase". Algorithm I states all the steps required to arrange the selected test cases T' into an order that will increase the rate at which T' detect faults. The steps are:

Encode the solution: The encoding technique used is permutation encoding as one of the techniques used in encoding the initial solution. To encode the solution T` = {t1 t2 t3 t5 t6 t7 t8}, T` = {1 2 3 5 6 7 8}.

Initial population generation: A random population of n of T' is generated as the initial population. Assuming random number is the size of T', the initial population is shown in Table 2.

Evaluate the fitness: Calculate the fitness value for each chromosome in the population using algorithm II. The fitness value of each chromosome in the population generated above is shown in Table 2. The selected test cases with their affected statements are given as:

- t1 = {n2 n3 n4}
- t2 = {n2 n3 n6 n7 n9}
- t3 = {n2 n4 n6 n10}
- t5 = {n9 n10}
- t6 = {n2 n3 n4 n9 n10 n13}
- t7 = {n10 n13}
- t8 = {n2 n3 n6}

Table 2. Fitness values of initial population

Number	Cromosome	Fitness
1	2 3 6 8 7 1 5	551.50
2	2 8 1 3 7 5 6	496.50
3	3 1 6 2 7 5 8	510.50
4	7 2 5 8 6 1 3	541.00
5	1 6 3 2 7 8 5	511.50
6	8 7 2 5 6 1 3	521.50
7	8 7 1 6 3 2 5	466.00
Total		3608.5

Table 3: Fitness values of new population after first iteration

Number	Chromosome	Fitness
1	2 3 6 8 7 1 5	551.50
2	3 1 6 2 7 5 8	510.50
3	7 2 5 8 6 1 3	541.00
4	1 6 3 2 7 8 5	511.50
5	8 7 2 5 6 1 3	521.50
6	6 3 2 7 5 8 1	560.50
7	3 2 5 7 6 8 1	526.50
Total		3723.0

To compute the fitness of chromosome 1 in table 2 using algorithm II: Chromosome 1 = {2 3 6 8 7 1 5}. Compute the fitness of each gene by calculating the fault severity by reducing the severity to 5% of the initial severity if the fault is already visited by the preceding test case and multiply it by the position of the gene in the chromosome, as shown below:

- Gene 2 = {n2 n3 n6 n7 n9} = 10+10+10+10+10 = 50*7 = 350 //None of the nodes was visited
- Gene 3 = {n2 n4 n6 n10} = 0.5+10+0.5+10 = 21*6 = 126 // nodes 4 and 10 are not visited by gene 2
- Gene 6= {n2 n3 n4 n9 n10 n13} = 0.5+0.5+0.5+0.5+0.5+10 = 12.5*5 = 62.5 //node 13 neither visited by gene 2 or gene 3
- Gene 8 = {n2 n3 n6} = 0.5+0.5+0.5 = 1.5*4 = 6
- Gene 7 = {n10 n13} = 0.5+0.5 = 1*3 = 3
- Gene 1 = {n2 n3 n4} = 0.5+0.5+0.5 = 1.5*2 = 3
- Gene 5 = {n9 n10} = 0.5+0.5 = 1*1 = 1

The fitness of the chromosome will be 350+126+62.5+6+3+3+1 = 551.5. The remaining chromosomes are computed using the same process.

Selection: Two (two chromosomes) are selected from the population for crossover based on their fitness calculated using algorithm II. The fitter individuals have higher chance of being selected for next generation. As shown in Table 2, chromosomes 1 and 4 are the best individuals among the population and have the maximum chance of being selected for crossover: Chromosome 1 = P1 = {2 3 6 8 7 1 5} and Chromosome 4 = P2 = {7 2 5 8 6 1 3}.

Crossover: After selection of the two parents, crossover operation is applied to the selected chromosomes. It involves swapping of genes or sequence of bits in the string between two individuals.

A valid solution would need to represent a child where every test case/gene is included at least once and only once. The authors use ordered crossover to produce valid child for next generation. Generate a random integer number r, between 1 and (numberOfGenes-1) to be the crossover point. A subset of the first parent (P1) containing first rth genes are copied and added to the first child. Genes/test cases which are not yet in the first child are added from the second parent in their order. The second child will contain the first rth genes from the second parent (P2) and the remaining genes from the first parent (P1).

- Offspring = Child1 and Child2 = orderedCrossover (Pi, Pj) = orderedCrossover (P1, P2)
- P1 = {2 3 6 8 7 1 5} and P2 = {7 2 5 8 6 1 3}
- If r = 3,
- Child1 = C1 = {2 3 6 ...} from P1, then 7 5 8 1 from P2 and
- Child2 = C2 = {7 2 5 ...} from P2, then 3 6 8 1 from P1
- Therefore,
- C1 = {2 3 6 7 5 8 1} and C2 = {7 2 5 3 6 8 1}

Mutation: The mutation operation also needs to be adjusted, so that it will not add random test case/gene to the offspring, possibly causing a duplicate. The authors use swap mutation to avoid duplication of test cases. Two random integer numbers between 1 and (numberOfGenes – 1) are generated, i.e. r1 and r2. The genes of these positions will simply swap their genes for the first child (C1), and the same process is repeated for the second child (C2):

- C1 = Swap mutation (child1)
- C2 = Swap mutation (child2)

Given:

- C1 = {2 3 6 7 5 8 1} and C2 = {7 2 5 3 6 8 1}
- If the two random numbers are 1 and 3, then the first child will be
- C1 = {6 3 2 7 5 8 1}

If also the two random numbers for the second child are 1 and 4, then second child will be: C2 = {3 2 5 7 6 8 1}.

Evaluate the fitness: Evaluate the fitness of the two offspring C1 and C2 using algorithm II.

Replacement: C1 = 560.5 and C2 = 526.5. Add the two offspring to the population and remove two worst chromosomes from the new population. After the first iteration, Table 2 will becomes Table 3.

For the next iteration, chromosomes 1 and 6 are the best individuals among the chromosomes in Table 3 and have the maximum chance of being selected for crossover: Chromosome 1 = P1 = {2 3 6 8 7 1 5} to be parent1 and Chromosome 6 = P2 = {6 3 2 7 5 8 1} to be parent2.

Termination: Check if the termination condition is not true, repeat steps 4-9, else end the process and return the optimal ordering.

Algorithm 1: To prioritize the selected test cases:

- ModGAPriorTCase (T', EvaPrioTcase) {
- EvaPrioTcase: is the set of best ordered test cases
- Te: is the set of encoded position of selected test cases as the solution
- Pc: is the set of chromosomes generated as population from Te
- F(Pi): is the fitness of chromosome i
- //Encode the selected test cases (T') using permutation encodintion 7 Te = {i_1, i_2, … , i_p)
- Generate n random chromosomes as initial population
- //Evaluate the fitness of each chromosome using
- algorithm II
- For each chromosome 1 to n
- F(P_i) = calcFitness(chromosome)
- end for
- Select two chromosomes from the population for crossover based on their fitness
- REPEAT
- Perform crossover
- Perform mutation on the offspring
- //evaluate the fitness of the two offspring using algorithm For each chromosome 1 to 2
- F (P_i) = calcFitness (chromosomei)
- end for
- add the two offspring to the population

Algorithm 2: For computing fitness of chromosome

```
calcFitness (chromosomei)    {
    gene: is a test case in the selected test cases
    allele: is a node in the test case coverage information
    alVisited: are the alleles already executed
    For each gene j in the chromosome i
        For each allele a in the gene j
            If a is already executed
                assign reduce fault severity
            otherwise
                assign initial faultseverity
            end if
            add a to the alVisited
            compute the total severities of all the alleles
        end for
        compute the fitness of the gene j
    end for
    sum the fitness of all the genes for chromosome i.
    return the fitness
end calcFitness
```

Experimental design: The researcher conduct experiments to evaluate the rate at which faults are detected for the computing fitness value of GA using equal severity and reduced severity of fault to 5, 10, 50 and 75% in regression test case prioritization of selected test cases, and run the experiments on the same computer using JAVA jdk1.7 with Neatbeans IDE on Intel ® Core™ i5-3470 at 3.2GHz and 8 GB RAM, under Microsoft Window 7 Professional.

Goal of the study: The goal of our study is to evaluate the effectiveness of computing fitness value based on fault severity reduction of previously executed statements for regression test case prioritization in using genetic algorithm. This research will be important to software testers in ordering the selected test cases for regression testing. Our research question is:

RQ1. How effective is the computation of fitness value based on reduction in faults severity of statement/fault executed by the preceding test cases in using genetic algorithm to prioritize selected test cases?

To address the question above, experiments were conducted to evaluate the performances based on rate at which faults are detected. Based on the above research question, the following hypothesis is presented:

The null hypothesis 1 (H_{0rft}) = There is no significant difference in the mean of rate of fault detection in using genetic algorithm to prioritize test suite using the same severities (SS) and using different reduced severity (Rd = 5%, Rd = 10, Rd = 50 and Rd = 75%) to compute fitness value in GA to prioritize selected test cases, and can be formulated as: $H_{0rft} = \mu_{R5} = \mu_{R10} = \mu_{R50} = \mu_{R75} = \mu_{SS}$ Where μ_j is the mean rate of faults detection scores of technique j measured on the nine programs.

Alternative hypothesis 1 (H_{1rft}) = There is a significant difference in the mean of rate of fault detection in using same severity (SS) and different reduction of fault severity (Rd = 5%, Rd = 10, Rd = 50 and Rd = 75%) to compute fitness value of GA to prioritize selected test cases. It can be formulated as: $H_{1rft} = \mu_{R5} \neq \mu_{R10} \neq \mu_{R50} \neq \mu_{R75} \neq \mu_{SS}$ (at least one of the means is different from the others)

Experimental setup: The empirical procedure for our proposed approach is: given a set of selected test cases, coverage information for each test case and affected statements, prioritize the selected test cases using GA with different fitness functions and compute the APFD of the prioritized test cases.

Three programs with three different versions each were used for empirical evaluation of our proposed approach which make up nine (9) programs; the Cruise Control (CC) is from a Software-artifact Infrastructure Repository (SIR) by Do *et al.* (2005); a repository that provide software for experimentations, binary search tree (BS) is from sanfoundry technology education blog by Bhojasia and ATM machine (AT) by Deitel and

Table 4: Summary of the sample applications

SPr	LOC	NC	NM	NT	Nmt
CC1	283	4	33	10	36
CC2	310	5	36	13	84
CC3	339	6	38	15	167
BS1	293	3	25	17	52
BS2	323	4	28	20	43
BS3	343	5	30	22	119
AT1	670	12	42	23	49
AT2	740	13	47	26	120
AT3	770	14	50	28	191

Paul (2005) is a case study provided in Java how to Program book for the implementation of the ATM machine, as shown in Table 4; SPr is the sample program, LOC is the lines of code, NC is the number of classes for each version, NM is the number of methods, NT is the number of test cases, NMt is the number of mutants, CC1, CC2 and CC3 are the Cruise control program versions 1, 2 and 3 respectively, BS1, BS2 and BS3 are binary search tree program versions 1, 2 and 3 respectively, and AT1, AT2 and AT3 are the ATM program versions 1, 2 and 3 respectively.

At CC1, brake () method and part of handleCommand() method are mutated to have 36 mutants. At CC2 methods accelerate (), engineOff (), and part of handleCommand () are mutated in addition to methods mutated in CC1 to have 84 mutants. At CC3 the methods engineOn (), enableControl (), disableControl (), part of handle Command () are mutated and the constructor CarSimulator () deleted together with the mutants from version 2 to have 167 mutants. At BS1, the methods countNode () and part of delete () and also some class-level mutants are mutated to have 52 mutants. At BS2, the methods isEmpty () and part of delete () are mutated to have 43 mutants. At BS3 the methods insert and search are mutated and also some class-level mutants to have 119 mutants. At AT1, the method execute () of deposit and balanceInquiry together with their constructors and the method credit () of bank database and account classes are mutated to have 49 mutants. At AT2 in addition to mutants in version 1, methods isSufficient Cash (), dispenseCash () of class CashDispenser with its constructor, debit () of class Account and part of the method execute () of class Withdrawal are mutated and some class-level mutants to have 120 mutants. At AT3 the method validatePin () of class Account with its constructor, the method authenticateUser () of class BankDatabase with it constructor, and part of the method execute () of class Withdrawal are mutated and some class-level mutants together with mutants in version 2 to have 191 mutants.

RESULT AND DISCUSSION

The data collected from the experiments are shown in Table 5. Table 5 gives the results of the nine (9) programs, Where SPr is the sample program, APFD is a

Table 5: Summary of the APFD of the three programs with 3 versions each

Sample program	APFD using different reduced severity				
	Rd = 5%	Rd = 10%	Rd = 50%	Rd = 75%	SS
CC1	75.0	75.0	65.0	65.0	65.0
CC2	72.2	72.2	61.9	59.6	56.7
CC3	85.7	82.5	76.4	73.4	73.4
BS1	66.5	66.5	66.5	60.5	60.5
BS2	87.1	87.1	81.2	78.2	81.2
BS3	93.4	93.6	90.9	89.6	88.6
AT1	73.7	73.7	73.7	73.7	67.5
AT2	89.1	88.3	82.0	79.9	67.3
AT3	94.2	91.2	87.4	83.9	81.5

matric for measuring the effectiveness of test case prioritization, the four columns represent different values obtained using different fault severity reduction; Rd = 5%, Rd = 10, Rd = 50 and Rd = 75% and the last column represents values for using the SAME Severity (SS).

Figure 1 compares the percentages of faults detected over the programs. The horizontal axis represents the nine programs and vertical axis represents the percentages of faults detected for each reduction approach on each program. From the figure, the highest percentage of faults detected 75.0 and 72.2% are the same in Rd = 5% and Rd = 10% for CC1 and CC2 programs respectively, whereas for CC3 Rd = 5% scores 85.7% and Rd = 10% scores 82.5%. Rd = 50% accounted for the next highest, i.e., 65.0, 61.9 and 76.4% respectively, Rd = 75% has 65.0, 59.6 and 73.4%, then SS scores 65.0, 56.7 and 73.4%, respectively. For BS1 program, Rd = 5%, Rd = 10 and Rd = 50% showed identical results of 66.5%, while Rd = 75% and SS have least scores of 60.5%. The rate of fault detection for BS2 is also identical in Rd = 5% and Rd = 10% of 87.1% and BS3 are higher in Rd = 10% of 93.6% followed by Rd = 5% with 93.4% compared to Rd = 50% with score of 81.2% and 90.9%, Rd = 75% with 78.2% and 89.6% and SS scores 81.2 and 88.6%, respectively. In AT1, Rd = 5%, Rd = 10%, Rd = 50% and Rd = 75% show the same fault detection rate of 73.7%, while SS scores 67.5%. But in AT2 and AT3 Rd = 5% and Rd = 10% have the higher scores of 89.1 and 88.3%, respectively and Rd = 50% has the next scores of 82.0% followed by 79.9% for Rd = 75% and 67.3% for SS. At AT3, Rd = 5%, Rd = 10%, Rd = 50%, Rd = 75% and SS have scores of 94.2, 91.2, 87.4, 83.95 and 81.5%, respectively.

Figure 2 and 3 shows the mean of faults detected from all the application in the three approaches. The y-axis represents the mean percentages of faults detected and the x-axis represents the four reduction approaches. As shown in the figure, reduced prioritization with Rd=5% scored the highest mean scores of 81.9% and Rd = 10% scored second highest mean scores of 81.1%, followed by Rd=50% 76.1% while Rd=75 and SS have the lowest mean scores of 73.8 and 71.3%, respectively.

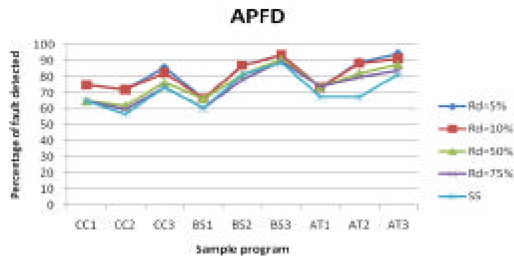

Fig. 2: Comparison of the fault detection rate across the programs

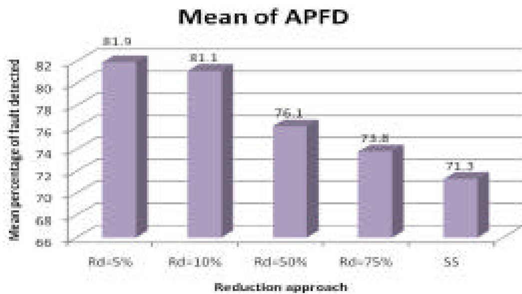

Fig. 3: Comparison of the average/mean of faults detected by each approach

From the above results, we can say that there is significant difference in the mean of rate of fault detection in using Same Severity (SS) and different reduction of fault severity (Rd = 5%, Rd = 10, Rd = 50 and Rd = 75%) to compute fitness value of GA to prioritize selected test cases.

Discussion: What we observed in the preceding section are discussed in this section. The results of our empirical study of the four reduction techniques (R = 5%, Rd = 10%, Rd = 50 and Rd = 75%) and SS drawn from the nine programs show that Rd = 5% is able to provide better results in term of APFD than the other four approaches.

From the results, the values from the CC1 and CC2 programs for Rd = 5% and Rd = 10% have identical scores and the highest rate of faults detection of 75.0% and 72.2% respectively, followed by Rd = 50%, Rd = 75 and SS for CC1 with 65, 61.9, 59.6 and 56.7% for CC2 respectively. This might be due to reduction of initial fault severity of a fault if executed by the previous test case to a small value 5 and 10% of the initial severity of fault compared to reduction to 50% and Rd = 75% where the initial value is only reduced to 1/2 and 3/4 respectively. Rd = 5% was found to have the highest scores in CC3 compared to Rd = 10 in CC1 and CC2 where they have identical values. This might be due to the higher number of mutants in CC3 compared with the number in CC1 and CC2. So also in other approaches, the values of APFD in CC3 are higher in CC1 and CC2. The performance of Rd = 5%, Rd = 10% and Rd = 50% obtained the same values of 66.5% while Rd = 75% and SS obtained the same values of 60.5%. Rd = 5% and Rd = 10% performed better in BS3 than in BS1 and BS2, so also the other three approaches. Rd = 5% and Rd = 10% performed better and identical in all the three versions of AT compared to the other three approaches.

From the results shown in Table 5, the four reduction techniques and the same severity perform better in version 2 than version 1 of the sample programs except for CC2 and CC1 in some approaches. This shows that the more the test cases and faults involved, the better the performance of the prioritization technique. So also version 3 of all the sample programs produced better results compared with other two versions. This also shows that the more test cases and faults, the better the performances of prioritization techniques. In general, it was observed that the more the percentages of reduction approach, the less the performances of the approach. Approach that does not reduce fault severity (SS) to any percentage performed poorly compared with the four reduction approaches.

This means that if the rate of fault detection is to be considered in using genetic algorithm for regression test case prioritization of selected test cases, using approach with reduced fault severity to compute the fitness value would be better for regression testing.

CONCLUSION

A regression test case prioritization approach that ordered selected test cases T` using GA with reduction in fault severity when a statement is executed by the preceding test cases was proposed. Given a set of selected test cases that was identified based on the affected statements, the coverage information and affected statements, the proposed approach prioritize the selected test cases using genetic algorithm with fault severity reduction to 5% of the initial fault severity.

The effectiveness of the approach was evaluated using APFD metric. In this paper, three sample programs with three different versions each are used for the evaluation. From the results presented, Rd = 5% provides better results in the nine programs in term of APFD. Based on the measured performance obtained from the results, GA with reduced severity of fault prioritize selected test cases more effectively compared to using GA with the same severity of fault. The more the effectiveness of test

case prioritization technique, the better the technique and the less cost of regression testing. Our proposed approach is based on the codes of the software, which might be time consuming when applied to software that has very large number of LOC (line of codes) which becomes its limitation. Also, if the test cases are very small, there would be no need of ordering the selected test cases. Another limitation is that it was assumed that all test costs are the same and faults severities are uniform except if already executed by the preceding test case. As a feature work, a hybrid approach that will detects faults at high level of abstraction and code level to be used for larger programs will be propose.

REFERENCES

Ahmed, A.A., M. Shaheen and E. Kosba 2012. Software testing suite prioritization using multi-criteria fitness function. Proceedings of the 22nd International Conference on Computer Theory and Applications (ICCTA), October 13-15, 2012, IEEE, Alexandria, Egypt, ISBN: 978-1-4673-2823-4, pp: 160-166.

Deitel, H.M. and D.J. Paul, 2005. Java How to Program. 6th Edn., Pearson Education, Upper Saddle River, New Jersey,.

Do, H., S. Elbaum and G. Rothermel, 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Eng., 10: 405-435.

Elbaum, S., A. Malishevsky and G. Rothermel, 2002. Test case prioritization: A family of empirical studies. IEEE Trans. Software Eng., 28: 159-182.

Gupta, R. and A.K. Yadav, 2013. Study of test case prioritization technique using APFD. Int. J. Comput. Technol., 10: 1475-1481.

Huang, Y.C., C.Y. Huang, J.R. Chang and T.Y. Chen, 2010. Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history. Proceedings of the 2010 IEEE 34th Annual Conference on Computer Software and Applications, July 19-23, 2010, IEEE, Seoul, Korea, ISBN: 978-1-4244-7512-4, pp: 413-418.

Kumar, H., V. Pal and N. Chauhan, 2013. A hierarchical system test case prioritization technique based on requirements. Proceedings of the 13th Annual International Conference on Software Testing, December 4-5, 2013, University of Science and Technolog, Bangalore, India,-pp: 4.

Musa, S., A.M. Sultan, A.B. Abd-Ghani and S. Baharom, 2014a. A regression test case selection and prioritization for object-oriented programs using dependency graph and genetic algorithm. Res. Inventy: Int. J. Eng. Sci., 4: 54-64.

Musa, S., A.M. Sultan, A.B. Abd-Ghani and S. Baharom, 2014b. Regression test case selection and prioritization using dependence graph and genetic algorithm. Int. Organiz. Sci. Res.-J. Comput. Eng., 16: 38-47.

Purohit, G.N. and A.M. Sherry, 2014. Test suites prioritization for regression testing using genetic algorithm. Int. J. Emerging Technol. Comput. Appl. Sci., 14: 255-259.

Raju, S. and G.V. Uma, 2012. Factors oriented test case prioritization technique in regression testing using genetic algorithm. Eur. J. Sci. Res., 74: 389-402.

Raperia, H. and S. Srivastava, 2013. An empirical approach for test case prioritization. Int. J. Sci. Eng. Res., 4: 1-5.

Rothermel, G., R.H. Untch, C. Chu and M.J. Harrold, 2001. Prioritizing test cases for regression testing. IEEE Trans. Software Eng., 27: 929-948.

Singh, K. and P. Kaur, 2014. Efficient test cases of regression testing using genetic algorithm. Int. J. Adv. Res. Comput. Commun. Eng., 3: 7504-7506.

Xu, Z., Y. Liu and K. Gao, 2013. A novel fuzzy classification to enhance software regression testing. Proceedings of the 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), April 16-19, 2013, IEEE, Singapore, Asia, pp: 53-58.

You, L. and Y. Lu, 2012. A genetic algorithm for the time-aware regression testing reduction problem. Proceedings of the 2012 Eighth International Conference on Natural Computation (ICNC), May 29-31, 2012, IEEE, Chongqing, China, ISBN: 978-1-4577-2130-4, pp: 596-599.