International Journal of Soft Computing 10 (6): 359-368, 2015
ISSN: 1816-9503
© Medwell Journals, 2015

The Effects of Replacement Strategies of Genetic Algorithm in Regression Test Case Prioritizatoin of Selected Test Cases

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: Regression testing is one of the software maintenance activities that is time consuming and expensive. Design-based regression testing approaches have been proposed to address changes at higher levels of abstraction, these approaches may not detect changes in the method body and several of the code based addresses procedural programs. This study presents an optimized regression test case prioritization of selected test cases for object-oriented software using Genetic algorithm with different replacement strategies. The goal is to compare different replacement strategies of GA and select the best strategy that will make prioritization to order selected test cases based on their fitness. We provide case studies to demonstrate the differences between the strategies. We measured the performances of each replacement strategy in GA by using Average Percentage of rate of Faults Detection (APFD) metric. It was observed from the results that replacement of worst individual and replacing the parent increased the effectiveness of regression testing compared with two other replacement strategies in term of rate of fault detection.

Key words: Regression testing, Evolutionary algorithm, regression test case prioritization, GA replacement strategy, Malaysia

INTRODUCTION

According to Pressman (2000), software maintenance is one of the software testing activities in software engineering and is an expensive stage account for nearly 60% of the total cost of the software production. One of the most important activities in software maintenance is regression testing. Regression testing is performed in order to ensure that modifications due to debugging or improvement do not affect the existing functionalities and the initial requirement of the design. Based on these researchers regression testing takes almost 80% of the overall testing budget and up to 50% of the cost of software maintenance.

Regression testing is also defined as a software testing activity carried out to ensure that modifications made in the fixes or any enhancement changes is not impacting the previously working functionality. It is normally performed after enhancement or defect fixes in the software or its environment. The testing team needs a set of test cases to retest whenever, the defect fixes are made to be run to verify the defect fixes. There is also need to carry out an impact analysis to find out what areas may get impacted due to those defect fixes that resulted in identifying more test cases to take care of the impacted areas.

It is necessary to perform regression testing when:

- Change is in requirements and the code is modified according to the requirement
- Defect fixing
- The new feature is added to the software

The regression test selection technique will help in selecting a subset of test cases from the test suite. The easiest way in regression testing is the tester simply executes all of the existing test cases to ensure that the new changes are harmless and is referred as retest-all method (Jin *et al.*, 2010). Retest-all is the safest technique but it is costly unless if the test suite is small in size. The selected test case can be ordered at random but most of the test cases ordered first randomly can result in checking small parts of the modified software. Techniques that prioritize selected test cases will be an alternative approach.

Some code-based regression test case selection and prioritization were proposed by researchers to order the test cases. The code-based approaches are safe and easier to make in generating the model of the software. The prioritization of selected test cases to be used in the testing of modified program means reduction in the cost associated with regression testing.

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

An approach for cost-cognizant prioritization technique that orders test cases according to their historical information was proposed by Yu-Chi. The approach prioritizes test cases according to their various test costs and fault severities by analyzing the historical records and genetic algorithm. Researchers used historical information instead of code coverage information which provides details of the changes. A technique that identifies test cases that execute the changed lines of the source code and select them after deletion of lines from the execution history of the test cases was proposed by Malhotra *et al*. (2010) as a test case selection and prioritization technique. This proposed technique is applied to only small program with small lines of code. Lin *et al*. (2012) proposed a modification-revealing test case reduction approach that selects test cases which are used as substitutes for fault-revealing test cases.

Approaches that prioritized test cases based on rate of faults detected and impact of the faults were proposed by Gupta and Yadav (2013) and Raperia and Srivastava (2013) while Panigrahi and Mall (2014) approaches were based on analysis of dependence model. For the latter, researchers constructed a dependence model of a program from its source code and when the program is modified, the model is updated to reflect the changes. The affected nodes are determined by performing forward slices using the changed nodes as slicing criterion. The test cases that covered the affected nodes are selected for regression testing. The selected test cases are then prioritized by assigning initial weights. The weights are used as bases for prioritization which may have direct impact on the selection.

A metric was proposed by Qusef *et al*. (2014) to measure effectiveness of test case prioritization in regression testing and also a prioritization technique which can be used to improve the rate of dependency detection for regression testing. A source code and concept based test to code traceability hunter was presented by Kayes; the approach first exploits dynamic slicing to identify the set of candidate tested classes and next the external and internal textual information is used to discriminate between actual tested classes and helper classes.

A prioritization technique using genetic algorithm to prioritize the regression test suite was proposed by Kaur and Goyal (2011) based on complete code coverage. The effectiveness of the algorithms was evaluated using Average Percentage of Code Covered (APCC) metric. But, the technique used test suite instate of selected test cases that contained only test cases that are faults-revealing. In our previous regression testi case selection and prioritization approaches (Musa *et al*., 2014a, b), we presented an evolutionary approach using genetic algorithm but the changes were done manually which may result in making bias selection and the initial population in GA is only two individuals which reduce the possibility of forming best ordering.

In this study, we conduct studies to determine the use of four different replacement strategies in GA in order to choose among them. Knowledge of this parameter will help identify among the four strategies that will reduce the cost of regression testing by increasing the rate of fault detection and reducing the number of test cases to be used in testing the modified program when using GA to prioritize selected test cases.

MATERIALS AND METHODS

Code-based regression testing technique consists of three phases: the first involves detecting changes between the original program and modified program; the second involves selecting test cases based on the identified changes; the third is arranging the test cases based on the fault coverage.

Identify changes: The changes between the original program codes P and the modified program P' are identified in this step by analyzing the source code of the software. We used Mujava tool by Ma to mutate the source codes of the application. Mujava mutates codes at class-level and traditional-level. The latter is used in order to design method-level mutation operators.

Class level mutants operators: The class mutation operators are classified into four groups; based on the language features that are affected. Three of the groups are common to all object-oriented based on language features. The fourth group is specific to Java programming features. The four groups are: Encapsulation, Inheritance, Polymorphism and Java-Specific Features and some of these changes are shown in the source codes as Fig. 1. Figure 1 is a modified program with the changed statements bolded. The ESDG is presented in Fig. 2.

Changes: The changed statement s8 that is int c = 2 being changed to static int c = 2 where the word static is added making the variable c not been accessed directly. As shown in Fig. 2, the statement s26 is data dependent on the changed statement s8, therefore, s26 is affected

```
ce1. Public class Transaction  {
s2. public Transaction (int userAccountNumber, Screen
atmScreen,
         BankDatabase atmBankDatabase)        {
s3.      accountNumber = userAccountNumber;
s4.     screen = atmScreen;
s5.      bankDatabase = atmBankDatabase;
   }  // end constructor transaction
s6.     abstract public void execute( );
}   // end class transaction
e7. Class Withdrawal extends Transaction {
s8.        static int c=2;   // static added
s9.        Public int d;  // initial value deleted
s11.       public Withdrawal ( ) {
s12.            super( );   }
s13.       public void execute()    { ……………   }
s14.       public int getX( )  {   s15.   return x; }
s16.       public int getY( )  {   s17.    return y;  }
     ………………………….. } // end class withdrawal
ce20. Class Balance extends Transaction {
e21.      // public Balance( ) {    s22.      super( );     }  //
constructor deleted
e23.        public void execute()    {   ……………     }
e25.        void Add (int a) { s26. a=a+c}
e28.        void Add (int a, int b) {
s29.             this.Add(a);  //replacing the body       }
     }  // end class balance
ce35. Class Deposit extends Transaction {
e36.     public Deposit(  ) {
s37.         //  super( );       deleted super call        }
e38.    public void execute()    {
s40.        Withdrawal.getY( );  // Instate of getX( )
s41.       Balance.Add(x, y);
            }  //end method execute
}
ce42. Class PrePaidReload extends Transaction {              //
added class
 e43.    public PrePaidReload(  ) {
s44.        super( );       }
s45.    public void execute()    {  ……………  }
}   // end class prepaidreload
```

```
ce50. public class ATM {
e51.  private void performTransaction()    {
s53.        while (!userExited)
           {
s54.           switch (mainMenuSelection)       {
             //user chose to perform
s55.              case BALANCE_INQUIRY:
s56.              case WITHDRAWAL:
s57.              case DEPOSIT:
S58.              case PrePaidReload:
             //INITIALIZE AS NEW OBJECT
s59.          currentTransaction  =
createTransaction(mainMenuSelection);
s60.             currentTransaction.execute();
s60.                break;
           }//end switch
        }//end while
   }//end method performTransaction
 //return object of specified transaction
e61.   private Transaction createTransaction(int type)     {
        //determine which type of transaction to create
s62.        switch (type)
           {
s63.         case BALANCE_INQUIRY:
s64.            temp = new Balance(currentAccntNu, screen,
bankDbase);
s65.          case WITHDRAWAL:
s66.               temp = new Withdrawal(currentAccntNu, screen,
bankDbase, keypad, cashDispenser);
s67.           case DEPOSIT:
s68.              temp = new Deposit(currentAccntNu, screen,
bankDbase, keypad, depositSlot);
S69.          case PrePaidReload:
S70.             temp = new PrePaidReload(currentAccntNu,
screen, bankDbase, keypad, depositSlot);
          } //end switch
      s71.      return temp;
      }// end method createtransaction
  } // end class ATM
```

Fig. 1: Source code of a modified program

statement. The statement e21 is a deleted constructor for class balance that makes the instantiated statements of balance to use the default constructor of class balance. We identify all the statements that instantiate the object balance as changed statements and statements that depend on the instantiated object of balance type are affected statements. The statement s64 is changed statement and the statement s71 is affected statement since its dependent on s64 as shown in Fig. 2. The statement s29 which is a method overloading call replaced the body of another overloaded method, this makes the overloaded method e28 to be changed statement and all method calls to this overloaded method are affected statements.

Statement s37 is a deletion of super call statement that makes reference to user parent constructor declaration. The deletion of super statement makes reference to the default constructor of the parent class transaction. The statement e36 is changed statement and statement s68 as affected statement. The statement s40 is changed from Withdrawal.getX ( ) to Withdrawal.getY ( ) and is marked as changed statement. Statements that dependent on s40 are the affected statements. A class named prePaidRelod is added from statement ce42 to s45 that resulted in adding statements s58, s69 and the instantiated statement s70. Adding statements s58 and s69 and s70 make s59 and s71 as changed statements, respectively which make statement s60 to be affected.

Test case selection: We used algorithm 1 to select test cases that execute the affected statements identified in identify changes. If the following test cases have the following coverage information:

- t1 = {n1 n2 n3 n4}
- t2 = {n1 n2 n3 n6 n7 n9}
- t3 = {n1 n2 n4 n5 n6 n10}
- t4 = {n1 n12}
- t5 = {n1 n9 n10}
- t6 = {n1 n2 n3 n4 n9 n10 n13}
- t7 = {n10 n13}
- t8 = {n1 n2 n3 n6}
- t9 = {n1}
- t10 = {n1 n11}

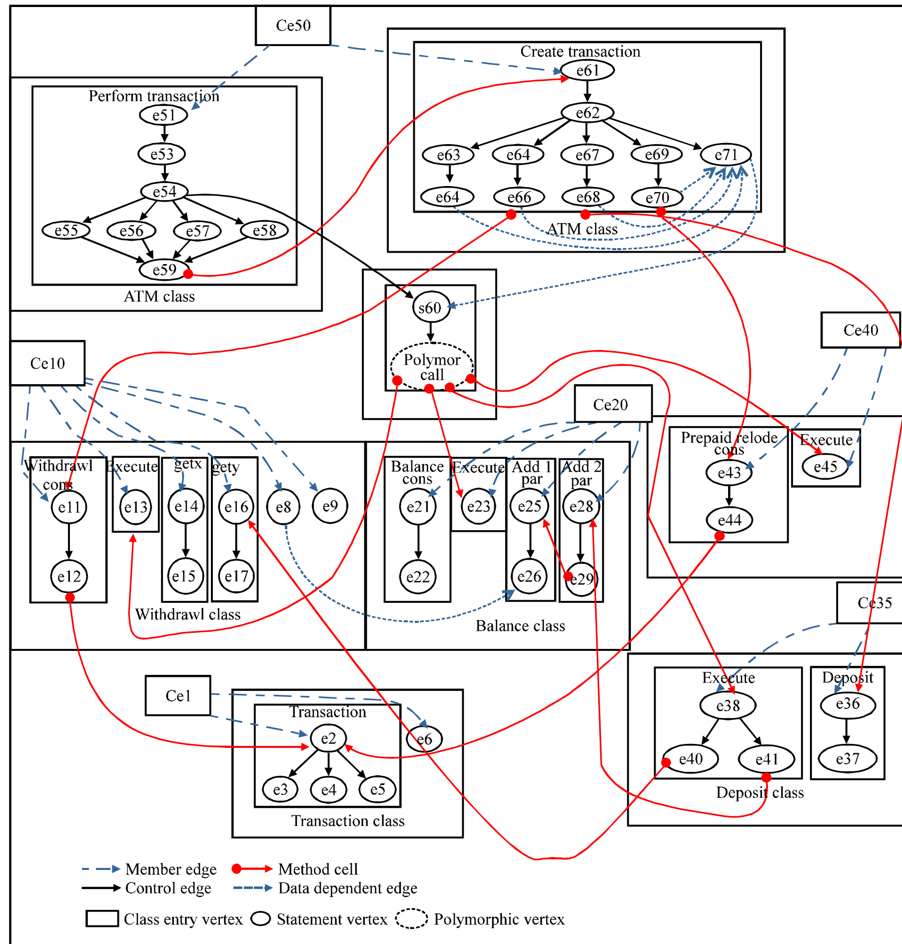

Fig. 2: ESDG for the source code of Fig. 1

Assuming the affected nodes/statements are n2 n3 n4 n6 n7 n9 n10 and n13 and then based on the algorithm 1, the selected test cases will be:

T' = {t1 t2 t3 t5 t6 t7 t8}

Algorithm 1:

EvolRegTCasePrior(M', T, CoveraInfo, Changed, AffectedNodes, T', EvaPrioTcase)

{
1. Changed: is the set of changed objects
2. M': is the updated extended system dependence graph
3. T = { t|t∈T}
4. Coverainfo: is the set of nodes covered by a test case
5. affectedNodes = { changed∪dependent nodes}
6. T' = { selected test cases}
7. EvaPrioTcase = {prioritized test cases}
8. affectedNodes = { ø }
9. T' = { ø }
10. For (node n : Changed) {
11. Find nodes that are dependent on node n (nDep)
12. affectedNodes = affectedNodes ∪ nDep
13. }
14. While (affectedNodes != Null) DO {
16. For (node a : affectedNodes) {
18. Find all test cases that cover node a (tcase)
19. T' = T' ∪ tcase
20. }
21. }
22. ModGAPriorTCase (CoveraInfo, T', EvaPrioTcase)
23. }

Test case prioritization: In this study, we intend to compare four replacement strategies for our proposed GA prioritization technique and is presented in algorithm 2. The replacement strategies are:

- Replacing the Worst individual (RW)
- Replacing at Random (RR)
- Replacing the Oldest individual (RO)
- Replace the Parent with the child (RP)

The following are the steps for the prioritization algorithm using genetic algorithm:

Encode the solution: We used permutation encoding as one of the encoding techniques used in encoding the initial solution. To encode the solution T' = {t1 t2 t3 t5 t6 t7 t8}, we have T' = {1 2 3 5 6 7 8}.

Initial population generation: We randomly generate a population of n individuals/chromosomes of T' as the initial population. Assuming the random number is 7, the initial population is shown in Table 1.

Evaluate the fitness: Calculate the fitness value for each chromosome using Eq. 1. The fitness values of the initial population are shown in Table 1:

$$Ft\,(pi) = \Sigma n(t_j) \times p_{ij} \quad \text{for } j = 1 \text{ to } k \tag{1}$$

Where:

- P_{ij} = The position of the individual/test case j in the parent/chromosome i
- $n\,(t_j)$ = The number of affected nodes in the test case j
- Ft (pi) = The fitness for chromosome i in the population n

Selection: Select two parents (two individuals) from the population. The selection depends on the strategy in use. Assuming the selected parents are chromosomes 5 and 7, regardless of the replacement strategy for crossover, then P1 = {2 5 1 8 3 6 7} and P2 = {2 1 3 7 8 6 5}.

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. We used ordered crossover to produce next offspring. A valid solution would needs to represent a child where every test case is included at least once and only once. We used ordered crossover to produce valid child for next generation.

Generate a random integer number r, between 1 and (numberOfGenes-1) to be used as crossover point. A subset of the first parent (P1) containing first rth genes will be selected 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 contains the remaining genes from the second Parent (P2) and the remaining genes of first Parent (P1).

Table 1: Fitness values of initial population

Chromosome	Fitness
7 5 3 8 2 6 1	92
2 1 5 7 6 8 3	99
2 1 7 3 6 8 5	105
2 8 7 1 6 3 5	103
2 5 1 8 3 6 7	100
8 3 1 7 2 6 5	97
2 1 3 7 8 6 5	104
Total	700

$$\text{child1} = \text{ordered crossover (P1, P2)} = c \times \text{parent1} + (1\text{-}c) \times \text{parent2}$$

$$\text{child2} = \text{ordered crossover (P1, P2)} = c \times \text{parent2} + (1\text{-}c) \times \text{parent1}$$

If the random number is 3, child1 will have {2 5 1} from parent1 and {3 7 8 6} from parent2. Therefore, child1 will be:

child1 = {2 5 1 3 7 8 6}

and child2 will be:

child2 = {2 1 3 5 8 6 7}

Mutation: We used swap mutation operation so that no random test case is added to the offspring, possibly causing a duplicate or introducing non-existent test case. Two random integer numbers were generated, i.e., r1 and r2. The genes of these positions will simply swap their genes for the first child (child1) and the same process repeated for the second child (child2).

Child1 = swapMutation (child1)

Child2 = swapMutation (child2)

C1 = {2 5 1 3 7 8 6}

If the two random numbers are 3 and 7, then child1 will be:

C1 = {2 5 6 3 7 8 1}

If also, the two random numbers for the second child are 2 and 5, then child2 will be:

C2 = {2 8 3 5 1 6 7}

Evaluate the fitness: Evaluate the fitness of the two offspring C1 and C2 using Eq. 1.

C1 = {2 5 1 3 7 8 6} = 108 and C2 = {2 8 3 5 1 6 7} = 104

Replacement strategies: As shown in algorithm 2, line 16 and line 17 (strikethrough and bolded) are the parts of our algorithm to be represented with different format at each strategy.

Replacing the Worst individual (RW): The replacement strategy selects the possibly best individual as the initial parents and after crossover and mutation operations, replaced the worst individual in the population with the offspring and selects the best offspring for the subsequent iterations. After the first iteration, Table 1 will be changed to Table 2 as the new population and the

Table 2: Fitness values of new population using RW

Chromosome	Fitness
2 1 5 7 6 8 3	99
2 1 7 3 6 8 5	105
2 8 7 1 6 3 5	103
2 5 1 8 3 6 7	100
2 1 3 7 8 6 5	104
2 5 1 3 7 8 6	108
2 8 3 5 1 6 7	104
Total	723

individuals that have a higher chance of being selected for the next iteration are chromosomes 2 and 6. After all the iterations, the best individual is returned.

Replacing at Random (RR): In this replacement strategy, the initial two parents are selected at random from the population and after crossover and mutation operations individuals are replaced at random with the offspring. If the two random integer numbers for replacement are 3 and 5, after the first iteration, the new population of Table 1 is shown in Table 3. Two random integer numbers are also generated to be used for the selection of the parents to be used for subsequent iterations. If the two numbers are 2 and 4, chromosomes 2 and 4 will be selected for the next iteration. After all the iterations, a random individual is selected and returned.

Replacing the Oldest individual (RO): In this replacement strategy, best individuals are used as initial parents and after crossover and mutation operations the oldest individuals are replaced with the offspring and the best individuals are selected as parents for the subsequent iteration. If the oldest individuals are chromosomes 1 and 2, then the new population of Table 1 is shown in Table 4 and the individuals that have a higher chance of being selected for the next iteration are chromosomes 1 and 3. After all the iterations, the best individual is returned.

Replace the Parent with the child (RP): In this replacement strategy, best individuals have the best chance of being selected as the parents (P1 and P2) and after crossover and mutation operations, the worst parents are replaced with the offspring and one of the offspring and the best individual are used as parents for the subsequent iterations. After the first iteration, Table 5 will be the new population of Table 1. Chromosome/individual 3 together with the child1 will be used as parents for the next generation. After all the iteration, the best among the offspring is returned.

Termination: Check if the termination condition not true, repeat step selection to termination, else, end the process and return one individual. The value returned dependents on the replacement strategy as seen in each strategy.

Table 3: Fitness values of new population using RR

Chromosome	Fitness
7 5 3 8 2 6 1	92
2 1 5 7 6 8 3	99
2 5 1 3 7 8 6	108
2 8 7 1 6 3 5	103
2 8 3 5 1 6 7	104
8 3 1 7 2 6 5	97
2 1 3 7 8 6 5	104
Total	707

Table 4: Fitness values of new population using RO

Chromosome	Fitness
2 5 1 3 7 8 6	108
2 8 3 5 1 6 7	104
2 1 7 3 6 8 5	105
2 8 7 1 6 3 5	103
2 5 1 8 3 6 7	100
8 3 1 7 2 6 5	97
2 1 3 7 8 6 5	104
Total	721

Table 5: Fitness values of new population using RP

Chromosome	Fitness
2 5 1 3 7 8 6	108
2 1 5 7 6 8 3	99
2 1 7 3 6 8 5	105
2 8 7 1 6 3 5	103
2 5 1 8 3 6 7	100
2 8 3 5 1 6 7	104
2 1 3 7 8 6 5	104
Total	723

Algorithm 2:

```
ModGAPriorTCase (CoveraInfo, T', EvaPrioTcase)
 {
1.  Encode the selected test cases (T') using permutation encoding
2.  T' = { i1, i2, … …., ip)
3.        where i1, i2, ……., ip are the positions of test cases t1, t2, ….. tp
    in the test suite T
4.  Generate n random chromosomes P1, P2, … Pn of population from T'
    to be the initial population
5.     Each chromosome/parent Pi is the set of the selected test cases T'
6.  Evaluate the fitness of each individual using:
7.        Ft(pi) = Ft(pi) = Σ n(tj)×pij for k = 1 to n
8.  EvaPrioTCase = {ø}
9.  REPEAT {
10.       Offspring = child1 and child2 = orderedCrossover (Pi, Pj)
11.
12.     child1 = swapMutation ( child1 )
13.     child2 = swapMutation ( child2 )
14.     evaluate the fitness of the child1 and child2 using
15.            Ft(pi) = Ft(pi) = Σ n(tj)×pij for k = 1 to n
16.     //
17.     //
18.   } UNTIL maximum number of iterations
19.   return chromosome (EvaPrioTcase)
20.   If new test cases are added (T")
21.       EvaPrioTCase = EvaPrioTCase U T"
22. }
```

Goal of the study: The goal of our study is to evaluate the four replacement strategies in using GA to prioritize the selected test cases with the aims of comparing them in term of rate at which faults are detected. This research will

be important to researchers in selecting a replacement strategy when using GA to order selected test cases for regression testing. Our research question is:

RQ1: What is the most effective replacement strategy in using GA to prioritize selected test cases for regression testing?

To address the question above, we carry out experiments to assess the rate at which faults are detected in each strategy and evaluate its performance. Based on the above research question, we present the following hypothesis:

- The null hypothesis (H_0): there is no significant difference in the mean of rate of fault detection in using the four replacement strategies in using GA to prioritize selected test cases and can be formulated as: $H_0 = \mu_{ARW} = \mu_{ARR} = \mu_{ARO} = \mu_{ARP}$. Where, μ_j is the mean rate of faults detection scores of replacement strategy j in GA measured on the three programs
- Alternative hypothesis (H_1): there is a significant difference in the mean of rate of fault detection in using the four replacement strategies in using GA to prioritize selected test cases. It can be formulated as: $H_1 = \mu_{ARW} \neq \mu_{ARR} \neq \mu_{ARO} \neq \mu_{ARP}$ (at least one of the means is different from the others)

Experimental design: We present an experiment to evaluate the rate at which faults are detected for the four replacement strategies of GA in regression test case prioritization of selected test cases. We run the experiment on the same computer using JAVA with Neat beans IDE on intel®Core™ i5-3470 at 3.2 GHz and 8 GB RAM, under Microsoft Window 7 Professional. We also present the results of the experiment.

The empirical procedure for our proposed approach is: given a test suite (T) for the each program/application, i.e., CC (cruise control) (Do *et al*., 2005), BS (Binary Search Tree) and AT (ATM case study) (Deitel and Deitel, 2007), we used path-based integration testing to generate and save the coverage information for each test case t1, t2, ..., tn, construct extended system dependency graph for each program, use mujava to introduce changes into the programs, reflect the changes in the updated extended system dependence graphs, get all the affected statements/nodes that are dependent on the changes; the affected statements are statements that were affected directly by the modifications or affected as the result of dependence, determine which test cases in T are modification-revealing with respect to the affected statements, prioritize the selected test cases using GA with different replacement strategy and compute the APFD of the prioritized test cases using Eq. 2.

Table 6: The sample programs

SPr	LOC	NC	NM	NT	Nmt
CC	283	4	33	10	110
BS	293	3	25	17	76
AT	670	12	42	23	125

We used three programs for empirical evaluation of the four proposed replacement strategies; the Cruise Control (CC) is from a Software-artifact Infrastructure Repository (SIR) (Hyunsook, 2005); a repository that provide software for experimentations, Binary Search tree (BS) is from sanfoundry technology education blog (Bhojasia, 2013) and ATM machine (AT) is a case study provided in Java how to program book for the implementation of the ATM machine (Deital and Paul, 2005) as shown in Table 6; SPr is the sample program, LOC is the lines of code, NC is the number of classes for each program, NM is the number of methods, NT is the number of test cases, NMt is the number of mutants (changes). The changes were introduced using mujava.

We used mujava to mutate the source codes of the application. The mutants considered are class-level mutants, traditional-level mutants and addition of classes.

We compared four different replacement strategies in our proposed prioritization approach (GA) as shown in Table 7 where ARW is the average percentage of rate of faults detected using RW, ARR is the average percentage of rate of faults detected using RR, ARO is the average percentage of rate of faults detected using RO and ARP is the average percentage of rate of faults detected using RP. The APFD metric widely used in gauging the performance of program P and test suite T is given as:

$$APFD = 1 - \frac{Tf_1 + Tf_2 +, ..., + Tf_n}{nm} + \frac{1}{2n} \quad (2)$$

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 expose the faults 1, 2, ..., m

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

RESULTS

The data collected from the preceding section are shown in Table 7. For each replacement strategy (RW, RR, RO and RP) the table shows the percentage of faults detected for each sample application (CC, BS and AT). In order to be able to draw the conclusion we discuss results

Table 7: Results of the APFD of the three programs

SPr	ARW	ARR	ARO	ARP
CC	78.7	65.2	74.2	76.9
BS	87.0	76.8	76.2	85.1
AT	86.4	71.7	79.2	85.4

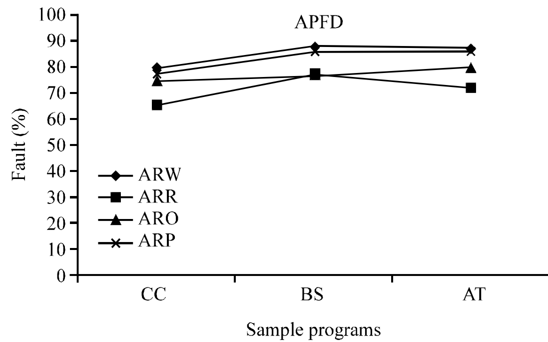

Fig. 3: Comparison of the fault detection rate across the programs

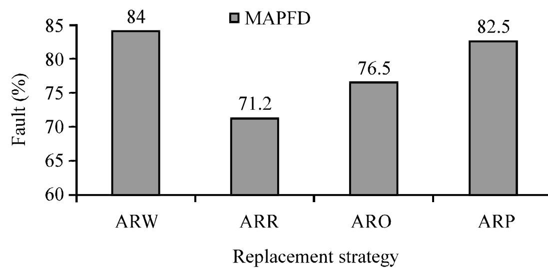

Fig. 4: Comparison of the average/mean of faults detected by each strategy

based on the rate at which faults are detected by using the four replacement strategies on three different applications and this is shown in Fig. 3. We also describe the average results of each selection strategy on each program as shown in Fig. 4.

Figure 3 compares the percentages of faults detected over the programs. The horizontal axis represents the three programs and vertical axis represents the percentages of faults detected for each replacement strategy of GA on each program. From the figure, the highest percentage of faults detected (78.7%) belonged to ARW for CC program whereas ARP accounted for the second highest, i.e., 76.9%. The ARR and ARO have 65.2 and 74.2%, respectively. For BS program, ARW and ARP showed almost identical faults detection rates of 87.0 and 85.1%, respectively while ARR and ARO also showed identical values of 76.8 and 76.2%, respectively. The rate of fault detection for ARW and ARP were identical rate in AT program of 86.4 and 85.4%, respectively higher than that of ARR (71.7%) and ARO (79.2%).

Table 8: Pairwise comparisons using t tests with pooled SD

Strategy	ARO	ARP	ARR
ARP	0.1532	-	-
ARR	0.1963	0.0174	-
ARW	0.0812	0.6879	0.0093

For the overall, the values of the three programs for ARW and ARP have the highest rate of faults detection scores. But, ARR is not consistent with the lowest scores in CC and third highest in BS.

Figure 4 shows the average of faults detected (MAPFD) from all the application in the replacement strategies. The y-axis represents the percentages of faults detected and the x-axis represents the four replacement strategies. As shown in the figure, ARW and ARP have the highest average scores of 84 and 82.5% respectively, while ARR and ARO have the lowest scores of 71.2 and 82.5%, respectively. These comparisons (Fig. 3 and 4) automatically provide that ARW and ARP showed a better performance results in faults detection in using GA. This means that there is difference in effectiveness between the four strategies. We apply statistical test to determine the level of significance of the differences.

Hypothesis testing (H_o): To determine the rejection or acceptance of null hypothesis, R statistical tool was used to conduct the test with a significance level value. The parametric ANOVA test was used with 5% significance level.

We obtained a F-value of 4.8575 which was greater than the critical value of 4.056 ($F_{crit} = F_{0.05(3,\ 8)}$) for the F-distribution at the degree of freedom of 3 and 8 and 95% confidence interval for the difference among treatments (strategies). Based on the decision rule, H_o is rejected if F-value>F_{crit} (4.8575>4.056) or if $p<\alpha$ (0.0328<0.05). With the respect of the above data, we reject H_o. From the analysis, we can see that there is a significant difference in mean rate of fault detection for the replacement strategies.

Since, there is a significant difference between the replacement strategies and ANOVA test cannot tell which specific strategy was significantly different from the others, we proceed to testing the main effect pairwise comparisons. To accomplish this, we apply pairwise.t.test() function to our independent variable (replacement strategies) and the result is shown in Table 8. The table shows that each strategy is compared with all the remaining strategies. From the table, the strategy ARW compared to the strategies ARO and ARR revealed low significant and very strong significant differences, respectively. Strategy ARP compared to the ARR revealed a significant difference. However, the results of the rest of comparisons gave no significant difference (value >0.1). This means that replacement

strategies ARW and ARP have overwhelming and strong evidences restectively to support the alternative hypothesis.

From the analysis above, ARW and ARP are significantly better than ARR but not significantly better than ARO. This analysis provided the evidence that a pair of treatment mean is not equal. Finally, we can say that ARW and ARP might have a better rate of fault detection compared to ARR.

DISCUSSION

In this study, we discuss what was observed in the preceding section. The results of our empirical study of replacement strategies (ARW, ARR, ARO and ARP) drawn from three programs showed that ARW strategy is able to provide considerably better results in faults rate detection than the other three replacement strategies. So, also ARP is better than ARO and ARR. All the three strategies are a little better than ARR, this may be due to the nature of random activity where no guides are given. The programs and their mutants are kept constant through out the use of different strategies. The replacement strategies on average performed better in BS and AT, showing that the strategies performed better on larger size program than the smaller size.

The results of the above experiment showed that there was statistically significant difference between the replacement strategies when using GA. ARW was found to be more effective in faults detection. This may be due to replacement of worst individuals during the iterations. ARR was found to be less effective in faults detection. This may be due to random replacement of the individuals which can result in replacing the fitter individuals during the iterations and selecting parents at random. ARP and ARW were found to be nearly identical. It may be due the nature of using one best offspring and fittest individual by ARP in every iteration as parents which is closely similar with ARW that used two fittest individuals in subsequent iterations as parents. This means that if the rate of fault detection is to be considered in using GA to order selected test cases, ARW would be better to be used as replacement strategy.

CONCLUSION

A regression test case prioritization approach that ordered selected test cases T' using GA with different replacement strategies was proposed. We used Mujava to mutate the original program. The approach used extended system dependence graph to identify changes at the statement level of source code of the original program and update the graph whenever the program is modified, store the changes in a file named changed and generate coverage information for each test case from the source code using path-based integration testing. The changed information is used to identify the affected statements from the updated model and select test cases that were identified based on the affected statements. The selected test cases were prioritized using genetic algorithm with different replacement strategies.

The effectiveness of the replacement strategy was evaluated using APFD metric. In this study, three sample programs are used for the evaluation. From the results presented, ARW provides considerably better results for rate of fault detection. Based on the measured performance obtained from the results, GA with ARW prioritized test cases more effectively compared to using GA with ARO and ARR which result in reducing the 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 as retest-all will be feasible. We also note that our proposed approach assumes that the ESDG are updated in a timely way, every time changes are introduced into the programs. This assumption also becomes a limitation of our approach. Another limitation is that we assumed that all test costs and faults severities are uniform.

As a feature study, we will present an approach that will compare other parameters in using GA to prioritize the selected test cases.

REFERENCES

Deitel, H.M. and P.J. Deitel, 2007. Java How to Program. 7th Edn., Pearson Education Inc., Prentice-Hall, New Jersey, ISBN: 0132222205, pp: 1596.

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.

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

Jin, W., A. Orso and T. Xie, 2010. Automated behavioral regression testing. Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST), April 7-9, 2010, Paris, France, pp: 137-146.

Kaur, A. and S. Goyal, 2011. A genetic algorithm for regression test case prioritization using code coverage. Int. J. Comput. Sci. Eng., 3: 1839-1847.

Lin, Y.D., C.H. Chou, Y.C. Lai, T.Y. Huang and S. Chung *et al*., 2012. Test coverage optimization for large code problems. J. Syst. Software, 85: 16-27.

Malhotra, R., A. Kaur and Y. Singh, 2010. A regression test selection and prioritization technique. J. Inform. Process. Syst., 6: 235-252.

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

Musa, S., A.M. Sultan, A.B. Abd-Ghani and S. Baharom, 2014b. 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.

Panigrahi, C.R. and R. Mall, 2014. A heuristic-based regression test case prioritization approach for object-oriented programs. Innovations Syst. Software Eng., 10: 55-163.

Pressman, R., 2000. Software Engineering: A Practitioners Approach. 5th Edn., Mcgraw Hill, USA.

Qusef, A., G. Bavota, R. Oliveto, A. De Lucia and D. Binkley, 2014. Recovering test-to-code traceability using slicing and textual analysis. J. Syst. Software, 88: 147-168.

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