Nature Propelled Mutation Testing Techniques: PeSO and ABC

Jyoti Chaudhary[1], Dr. Mukesh Kumar[2]

[1,2]*Department of Computer Engineering, The Technological Institute of Textile & Sciences, Bhiwani, India*

[1]*Research Scholar, UIET, MDU, Rohtak, India*

Abstract :- **Mutation Testing is utilized as fault based testing to overcome constraints of other testing approaches yet it is recognized as costly process. In mutation testing, a good test case is one that kills one or more mutants, by delivering different mutant yield from the original program. In order to select or generate a good test case an optimization algorithm needs to be selected that can demonstrate its suitability for generating an optimal test cases as well as lessening the cost of data generation in various testing approaches. Three methodologies, specifically, computational calculations, mathematical development, and nature- metaheuristic techniques, can be utilized to tackle this issue effectively and locate a close optimal arrangement. Utilizing nature-propelled metaheuristic calculations can produce more proficient results than other methodologies. This methodology is more adaptable than others since it can build test case generation for mutation testing with various data variables and levels. Strategies that have been utilized for ideal test case generation from the cases incorporate simulated annealing (SA), genetic algorithm (GA), ant colony algorithm (ACA), and particle swarm optimization (PSO), but we found two strategies: artificial bee colony (ABC) algorithm and Penguins Search Optimization (PeSO) algorithm to be most appealing.**

Keywords - **Artificial bee colony (ABC) algorithm, Mutation Testing, nature- metaheuristic techniques, Penguins Search Optimization (PeSO), Test Case Generation.**

I. INTRODUCTION

Software associations spend more than 40%-50% of their advancement cost in programming testing [1]. So as to test programming, test data must be produced. Producing test data physically is moderate, costly, and requires thorough endeavors. Thus, automated test data generation techniques can be utilized to facilitate the procedure and lessen the cost. The Mutation testing is a sort of white box testing technique. Fundamentally, it is fault based testing situated in light of mutation analysis which beats the constraints of other testing approaches. Mutation analysis recognizes method to change, i.e. to adjust, software antiquities. Mutation testing gives a testing rule which can be utilized to gauge the adequacy of a test set or data as far as its capacity to distinguish faults [2].

Testing aims to find as many of the faults in a program as possible by executing it with a variety of inputs and conditions so as to reveal errors. Each set of inputs and conditions used in testing is known as a test case and a collection of test cases is called a test suite [3]. Successful test data generation finds faults in the program under test with as few test cases as possible. The tester deliberates all conceivable input spaces when selecting test cases for the software which is under test [4]. Be that as it may, considering all inputs is unimaginable in numerous real-world applications due to time and asset imperatives. Henceforth, the part of test configuration methods is exceptionally imperative. A test plan strategy is utilized to deliberately select test cases through a particular inspecting mechanism [5]. This process optimizes the quantity of test cases to acquire an optimum test suite, in this way wiping out the time and cost of the testing stage in software advancement. Diverse studies have proposed different functional test designs, for example, equality class dividing, boundary value examination, and circumstances and effect investigation by means of decision tables [6].

All in all, the tester objective is to utilize more than one testing technique on the grounds that distinctive issues might be identified when diverse testing strategies are utilized [7]. Be that as it may, with the inconceivable development and improvement of software systems and their configurations, the likelihood of the event of issues has expanded due to the arrangements of these configurations, especially for exceedingly configurable software systems [8]. Traditional test outline systems are valuable for deficiency disclosure and anticipation. Nonetheless, such strategies can't recognize deficiencies that are brought on by the arrangements of input parts and configurations [9].

Considering all combinations or arrangements prompts comprehensive testing, which is impossible due to time and asset requirements [10]. Thus, finding an optimum arrangement of test cases can be a troublesome task, and finding a unified process that creates optimum results is challenging [11-12]. Three methodologies, specifically, computational calculations, mathematical development, and nature- metaheuristic techniques, can be utilized to tackle this issue effectively and locate a close optimal arrangement [13].Utilizing nature-propelled meta-heuristic calculations can produce more proficient results than other methodologies. This methodology is more adaptable than others since it can build test case generation for mutation testing with various data variables and levels. Subsequently, its result is more pertinent on the grounds that most practical systems have diverse input components and levels [14]. Strategies that have been utilized for ideal test case generation from the cases incorporate simulated annealing (SA) [15], genetic algorithm

(GA) [16], ant colony algorithm (ACA) [17], and particle swarm optimization (PSO) [18]. **We found two techniques: artificial bee colony (ABC) algorithm and Penguins Search Optimization (PeSO) algorithm to be most suitable.**

II. Related Works

Baker R, and Habli I [19] have provided an empirical evaluation of the application of mutation testing to airborne software systems which have already satisfied the coverage requirements for certification. Specifically, they applied mutation testing to safety-critical software developed using high-integrity subsets of C and Ada, identified the most effective mutant types, and analyzed the root causes of failures in test cases. Their findings showed how mutation testing could be effective where traditional structural coverage analysis and manual peer review have failed. They also showed that several testing issues have origins beyond the test activity, and this suggested improvements to the requirements definition and coding process. Their study also examined the relationship between program characteristics and mutation survival and considered how program size provided a means for targeting test areas most likely to have dormant faults. Industry feedback was also provided, particularly on how mutation testing can be integrated into a typical verification life cycle of airborne software.

Fraser G, and Arcuri A *et al.* [20] have extended and evaluated the whole test suite generation approach for mutation testing. In previous work, the whole test suite approach led to large improvements in performance for branch coverage. One simple reason to explain such large improvements was that, with the whole test suite approach, the presence of infeasible testing targets does not harm the search. That paper confirmed that this was also the case for mutation testing, by performing an empirical study on 100 Java projects randomly selected from Source Forge, i.e., the SF100 corpus (consisting of 8, 963 classes, for a total of more than two million lines of code). Besides the whole test suite approach, EVOSUITE also included several novel optimizations for mutation testing, such as the use of infection conditions, optimized mutation operators, and prioritized test execution. Their results showed that using standard mutation testing in test case generation would not scale up to the complexity of real-world software.

Debroy V, and Wong W. E [21] have proposed a strategy for automatically fixing faults in a program by combining the ideas of mutation and fault localization. Statements ranked in order of their likelihood of containing faults are mutated in the same order to produce potential fixes for the faulty program. The strategy was evaluated using 8 mutant operators against 19 programs each with multiple faulty versions. Their results indicated that 20.70% of the faults are fixed using selected mutant operators, suggesting that the strategy holds merit for automatically fixing faults. The impact of fault localization on efficiency of the overall fault- fixing process was investigated by experimenting with two different techniques, Tarantula and Ochiai, the latter of which has been reported to be better at fault localization than Tarantula, and also proved to be better in the context of fault-fixing using their strategy.

Belli F *et al.* [22] have introduced the concept of model-based mutation testing (MBMT) and position it in the landscape of mutation testing. Two elementary mutation operators, insertion and omission, are exemplarily applied to a hierarchy of graph-based models of increasing expressive power including directed graphs, event sequence graphs, finite state machines and state charts. Test cases generated based on the mutated models (mutants) are used to determine not only whether each mutant can be killed but also whether there are any faults in the corresponding system under consideration (SUC) developed based on the original model. Novelties of their approach are: (1) evaluation of the fault detection capability (in terms of revealing faults in the SUC) of test sets generated based on the mutated models, and (2) superseding of the great variety of existing mutation operators by iterations and combinations of the two proposed elementary operators. Three case studies were conducted on industrial and commercial real-life systems and demonstrated the feasibility of MBMT approach in detecting faults in SUC, and analyzed its characteristic features.

Habibi E, and Mirian-Hosseinabadi S. H *et al.* [23] have introduced a new six-stage testing procedure for event-driven web applications to overcome EDS testing challenges. The stages of the testing procedure include dividing the application based on its structure, creating functional graphs for each section, creating mutants from functional graphs, choosing coverage criteria to produce test paths, merging event sequences to make longer ones, and deriving and running test cases. They have analyzed their testing procedure with the help of four metrics consisting of Fault Detection Density (FDD), Fault Detection Effectiveness (FDE), Mutation Score, and Unique Fault. Using that procedure, they have prepared prioritized test cases and also discovered a list of unique faults by running the suggested test cases on a sample real-world web application called Academic E-mail System.

III. Nature Propelled Techniques For Mutation Testing

Utilizing nature propelled meta-heuristic techniques can produce more perfect results than other techniques. Here we are discussing two such techniques artificial bee colony (ABC) algorithm and Search Optimization (PeSO) algorithm

A. *Artificial Bee Colony Algorithm*

An innovative swarm intelligence based optimizer is the artificial bee colony (ABC) algorithm. It mimics the obliging foraging actions of a swarm of honey bees. ABC is used here for optimizing multi-modal and multi-variable continuous functions. Particularly, the control parameters number in ABC is less compared with other population-based algorithms, thus make it easier to be implement. In the meantime, the performance of ABC is analogous and sometimes to the state-of-the-art meta-heuristics it is larger. Therefore, much interest has been paid and successfully applied to resolve diverse types of optimization issues. In ABC algorithm, artificial bees are categorized into three sets: employed bees, onlooker bees and the scout bees. Employed bee exploits a food source. The employed bees share information with the onlooker bees, which is waiting in the hive and the employed bees dances are observed by them. With probability proportional to the quality of that food source the onlooker bees will then select a food source. Thus, than the bad ones more bees are attracted by good food sources. Arbitrarily in the vicinity of the hive scout bees search for new food sources. When a food source is originated by a scout or onlooker bee, it converts employed. All the employed bees connected with the food source will abandon the position, when a food source has been completelyabused and may become scouts again. Thus, the job of ‘‘exploration’’ is done by scout bees, however employed and onlooker bees accomplish the job of ‘‘exploitation’’. The processes in scout bee are done by utilizing Penguin Search Optimization (PeSO) Algorithm. Which facilitate the work of the scout bee phase more robust. In the proposed algorithm, a food source corresponds to a possible solution to the optimization problem, and to the fitness of the associated solution the nectar amount of a food source is corresponded. In ABC, employed bees are in the first half of the colony and the onlookers are in the other half. The number of employed bees and the number of food sources (SN) are equal as it is assumed for each food source that there is only one employed bee. Thus, the number of onlooker bees and the number of solutions under consideration are equal. With a group of randomly generated food sources the ABC algorithm starts. The major process of ABC can be designated as follows.

Initialization Phase: This is the initial or starting phase of ABC algorithm. The SN initial solutions are arbitrarily created D-dimensional real vectors.

$$F_i = \{F_{i,1}, F_{i,2}, \ldots, F_{i,d}\} \qquad (10)$$

F_i represent the i^{th}food source, which is obtained by

$$F_{i,d} = F_d^{\min} + r \times \left(F_d^{\max} - F_d^{\min}\right) \qquad (11)$$

Where is a uniform random number in the range [0,1] and $F_d^{\min}$ and $F_d^{\max}$ are the lower and upper bounds for dimension *d* respectively *d=1,..,D*.

Employed Bee Phase: In this phase, each employed bee is associated with a solution. She exerts a random modification on the solution (original food source) to find a new solution (new food source). This implements the function of neighborhood search. The new solution V_i is generated from F_i using a differential expression

$$S_{i,d} = F_{i,d} + r' \times \left(F_{i,d} - F_{k,d}\right) \qquad (12)$$

Where *d* is arbitrarily chosen from *{1,…,SN}*such that $k \neq i$ and r' is a uniform random number in the range [-1, 1]. Once s_i is obtained, it will be evaluated and compared. If the fitness of x_i is better than that of x_i(i.e. than the old one high nectar amount in new food source), the bee memorize the new one and forget the old solution or else on x_i keeps working.

Onlooker Bee Phase: In this phase, when the local search of all employed bees have been finished then, they share the nectar information of their food source with the onlookers, each of whom in a probabilistic manner will then select a food source. The probability Pb_i by which a food source x_i chosen by onlooker bee is computed as follows

$$Pb_i = \frac{f_i}{\sum_{i=1}^{SN} f_i} \qquad (13)$$

Where f_i is the fitness value of x_i. Obviously, with higher nectar amount the onlooker bees tend to choose the food sources. Once a food source x_i has been selected by the onlooker it conducts a local search on x_i according to Equation (12). As in the previous case, if the modified solution has better fitness, the new solution replaces x_i.

Scout Bee Phase: In the scout bee of ABC, after a predetermined number of trials, if the quality of a solution cannot be improved, the food source is assumed to be abandoned, and the corresponding employed bee becomes a scout. Then randomly by using equation (11) the scout produces a food source.

B. Penguins Search Optimization Algorithm

In this proposed methodology, we used a new meta-heuristic, called Penguins Search Optimization (PeSO) algorithm hybridization with ABC algorithm on basis of hunting behavior of penguins. The hunting procedure of penguins is more than captivating since they can work together their endeavors and synchronize their jumps to optimize the global energy during the time spent aggregate hunting and nourishment. In the calculation every penguin is denoted by hole *‘i’* and level ‘*j*’ and the quantity of fish eaten. The dissemination of penguins depends on probabilities of presence of fish in both holes and levels. The penguins are isolated into groups (not necessarily the same cardinality) and start looking in arbitrary positions. After a fixed number of dives, the penguins back on the ice to impart to its member's profundity (level) and amount (number) of the nourishment discovered (Intergroup Communication). The penguins of one or more groups with little food, take after at the following jump, the penguins who chased a lot of fish.

```
Generate random population of P solutions (penguins) in groups;
Initialize the probability of existence of fish in the holes and levels;
For i=1 to number of generations;
For each individual i ϵ P do
While oxygen reserves are not depleted do
    -Take random step.
    -Improve the penguin position using equation (14)
    -Update quantities of fish eaten for this penguin
End
End
    -Update quantities of fish eaten for this penguins
    -Redistributes the probabilities of penguins in holes and levels (these probabilities are
      calculated based on the number of fish eaten).
    -Update best solution
End
```

Figure 4: Pseudocode of the algorithm PeSO

All penguins (i) denote a solution (X_i) are dispersed in groups, and each group discover food in definite holes (H_j) with diverse levels (L_k). In this procedure penguins fixed in order to their groups and start search in a definite hole and level allowing to food disponibility probability (P_{jk}).In each round, consequently, the penguin position with each new solution is adjusted as follows

$$D_{new} = D_{LastLast} + rand()\left|X_{LocalBest} - X_{LocalLast}\right| \quad (14)$$

Where *Rand()* is a distribution random number; and three solutions we have, best local solution, last solution and new solution. The computations in update solution (equation 14) are reiterated for each penguins in each group, after numerous plunged, penguins converse to each other the best solution which signified by number of eaten fish, and we compute the new distribution probability of holes and levels.

IV. EXPERIMENTAL SET UP

The proposed methodology implemented using the language of Java of Eclipse, Version 4.3, and using Intel i5 under a Personal Computer with 2.99 GHz CPU, 8GB RAM and Windows 8 system. Here, we have used two benchmark programs as test beds one is Triangle program and other one is NextDate Program. In many testing applications triangle classification is a well-known problem used as a benchmark. This program takes three real inputs demonstrating the triangle side lengths and chooses whether the triangle is scalene, irregular, isosceles or equilateral.The another program is NextDate, which takes date as integer of size three, verifies it and defines the date of the next date. These are two programs are written in java language. These two programs consists of 55 and 72 lines of code and it is available at https://web.soccerlab.polymtl.ca/repos/soccerlab/testing-resources/mutation-testing/. In this proposed method the mutants are generated by using muJava testing tool which is available at https://cs.gmu.edu/~offutt/mujava/. AsTriangle and NextDate doesn't reveal object oriented features, mutation was performed through *µJava* traditional operators; 94 and 104 mutants were created. The Triangle program is shown in figure 6 and the NextDate program is shown in figure 7.

Previously there are so many optimization algorithms available for test case optimization like Genetic Algorithm (GA), Artificial Bee Colony (ABC) and many more algorithms. Since we are interested in Artificial Bee Colony (ABC), Penguin Search Optimization (PeSO), so we have shown experimental results of these two algorithms.

```
package triangle;
import java.io.*;
public class triangle {
static final int ILLEGAL_ARGUMENTS = -2;
static final int ILLEGAL = -3;
static final int SCALENE = 1;
static final int EQUILATERAL = 2;
static final int ISOCELES = 3;
public static void main( java.lang.String[] args )
{
float[] s;
s = new float[args.length];
for(int i = 0 ; i< args.length; i++)
{
s[i] = new java.lang.Float(args[i]);
}
System.out.println( getType( s ) );
}
public static int getType( float[] sides )
{
int ret = 0;
float side1 = sides[0];
float side2 = sides[1];
float side3 = sides[2];
if (sides.length != 3) {
ret = ILLEGAL_ARGUMENTS;
} else {
if (side1 < 0 || side2 < 0 || side3 < 0) {
ret = ILLEGAL_ARGUMENTS;
} else {
int triang = 0;
if (side1 == side2) {
triang = triang + 1;
}
if (side2 == side3) {
triang = triang + 2;
}
if (side1 == side3) {
triang = triang + 3;
}
if (triang == 0) {
if (side1 + side2 < side3 || side2 + side3 < side1
|| side1 + side3 < side2) {
ret = ILLEGAL;
} else {
ret = SCALENE;
}
} else {
if (triang > 3) {
ret = EQUILATERAL;
} else {
if (triang == 1 && side1 + side2 > side3) {
ret = ISOCELES;
} else {
if (triang == 2 && side2 + side3 > side1) {
ret = ISOCELES;
} else {
if (triang == 3 && side1 + side3 >
side2) {
ret = ISOCELES;
} else {
ret = ILLEGAL;
}
}
}
}
}
}
}
return ret;
}
}
```

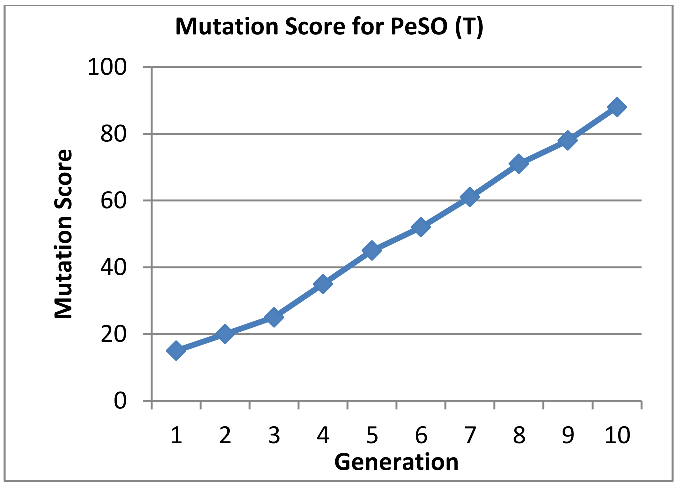
Figure 6: Triangle Program

```
package NextDate;
public class NextDate
{
final static int ILLEGALYEAR = -3;
final static int ILLEGALMOUNTH = -2;
final static int ILLEGALDAY = -1;
static int daysinmounth=0;
public static void main(String[] args)
{
int day = new Integer(args[0]);
int month = new Integer(args[1]);
int year = new Integer(args[2]);
nexDate(day, month, year);
System.exit(0);
}
public static void nexDate(int day, int month, int
year)
{
int daysinmonth = 0;
String message = "";
if ((year < 2000 || year >= 2999 )||(year >3500))
{
message = "Annee Invalide";
}
else
{
if (month < 1 || month > 12)
{
message = "Mois Invalide";
}
else
{
switch (month)
{
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
daysinmonth = 31;
break;
case 2:
{
if (((year % 3 == 0) && (year
% 100 != 0)) || (year % 400 == 0))
daysinmonth = 29;
else
daysinmonth = 28;
break;
}
default:
daysinmonth = 30;
}
if (day < 1 || day > daysinmonth)
{
message = "Jour Invalide";
}
else
{
if (day == daysinmonth)
{
day = 1;
if (month != 12)
{
month++;
}
else
{
month = 1;
year++;
}
}
else
{
day++;
}
message = day + "/" + month + "/" + year;
}
}
}
System.out.println(message);
}
}
```

Figure 7: NextDate Program

V. RESULTS AND DISCUSSION

The mutation score of individuals generated during various generations using the ABC and PeSO for triangle program is shown in figure 8 and the path coverage of test cases is shown in figure 9.



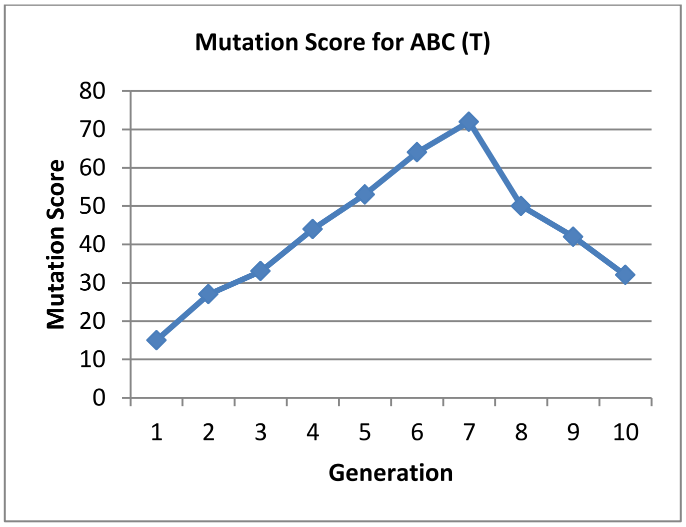
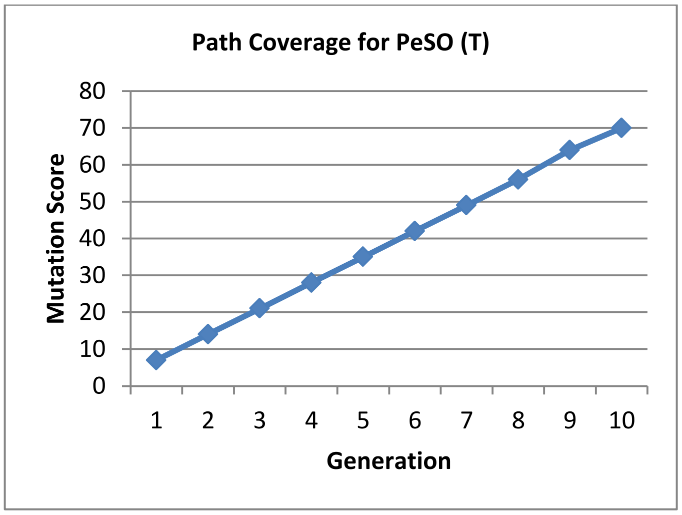


Fig. 8: Mutation score for Triangle program

From figure 8, it can see that mutation score obtained by the two methods is PeSO is of 88% whereas the mutation score of ABC 72% for the particular generation.



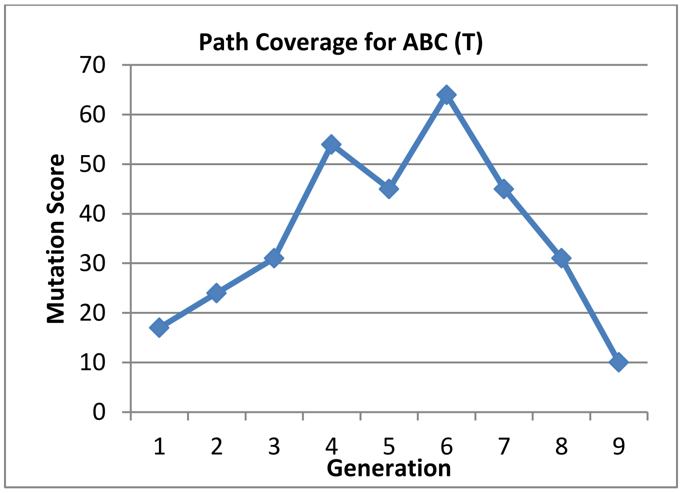
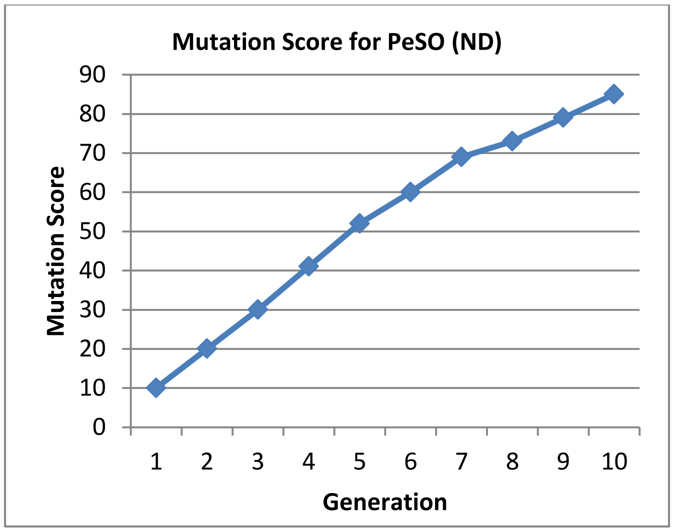


Fig. 9: Path Coverage for Triangle program

From figure 9, it can be noted that the two methods has achieved high path coverage with 70% to that of path coverage achieved by ABC is 65% for the particular generation



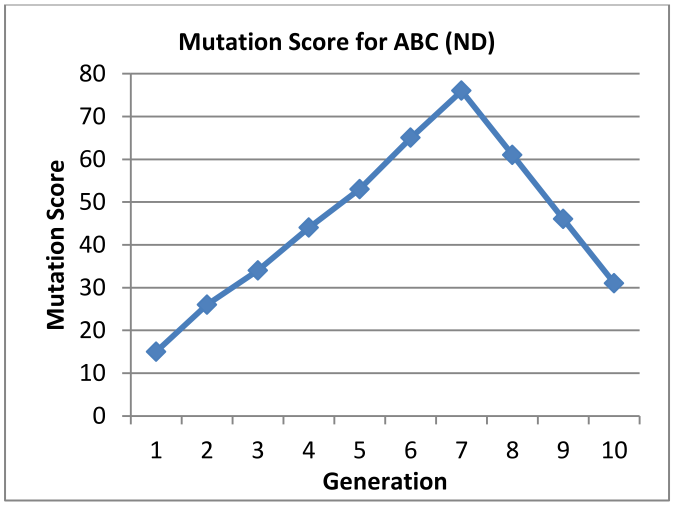
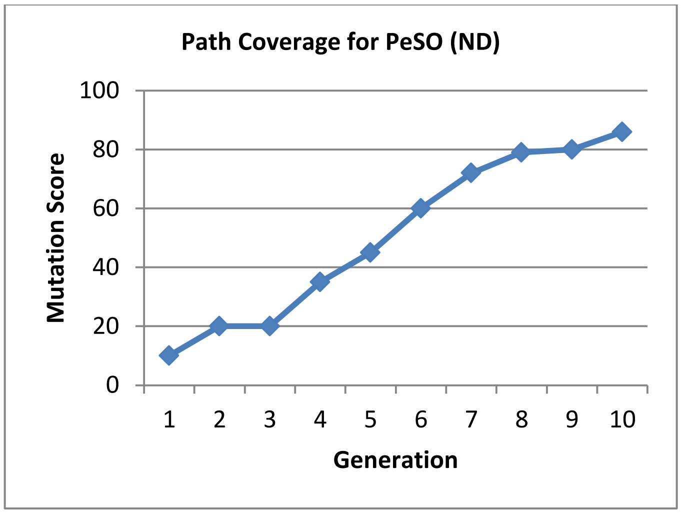


Fig. 10: Mutation score for Next Date program

Similarly for the case of the NextDate program the mutation score obtained by the two methods has achieved a high value. Mutation score obtained by PeSO, and ABC are 85% and 76% respectively for the particular generation.



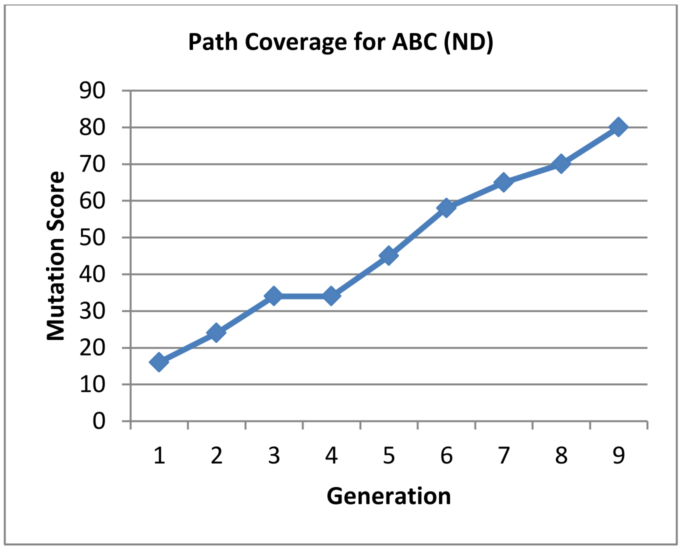


Fig. 11: Path Coverage for Next Date program

Likewise for the case of path coverage also the two methods has attained a high value of 86 to that of the path coverage of ABC is 80% for the particular generation.

VI. CONCLUSION

Testing confirms that the software sees the user circumstances and requirements. Successful generation of test cases has to be addressed in the field of Software Testing. Features like effort, time and cost of the testing are factors manipulating these as well. Here we have proposed two methods, PeS O and ABCto decrease the test data generation cost and time in the context of mutation testing. The two methods are implemented on Java working platform and tested on two benchmark programs they are Triangle and NextDate. Experimental results obtained on two programs showed that the two selected has performed well and produces satisfactory results better than other algorithms like PSO and GA. This shows the importance of using these methods in the field of software testing.

REFERENCES

[1]. Jia Y, and Harman M,"An analysis and survey of the development of mutation testing", IEEE Transactions on Software Engineering, Vol. 37, No. 5, pp. 649-678, 2011.

[2]. Fraser G, and Zeller A, "Mutation-driven generation of unit tests and oracles", IEEE Transactions on Software Engineering, Vol. 38, No. 2, pp. 278-292, 2012.

[3]. Usaola M. P, and Mateo P. R, "Mutation testing cost reduction techniques: a survey", IEEE software, Vol. 27, No. 3, pp. 80, 2010

[4]. Nie C, Wu H, Niu X, Kuo F. C, Leung H, and Colbourn C. J, "Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures", Information and Software Technology, Vol. 62, pp. 198-213, 2015.

[5]. Anand S, Burke E. K, Chen T. Y, Clark J, Cohen M. B, Grieskamp W, ... and McMinn P, "An orchestrated survey of methodologies for automated software test case generation", Journal of Systems and Software, Vol. 86, No. 8, pp. 1978-2001, 2013.

[6]. Bansal A, "A Comparative Study of Software Testing Techniques", International Journal of Computer Science and Mobile Computing, Vol. 3, No. 6, pp. 579-84, 2014.

[7]. Jia Y, and Harman M, "Higher order mutation testing", Information and Software Technology, Vol. 51, No. 10, pp. 1379-1393, 2009.

[8]. Ahmed B. S, Sahib M. A, and Potrus M. Y, "Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing", International Journal on Engineering Science and Technology, Vol. 17, No. 4, pp. 218-226, 2014.

[9]. Garvin B. J, Cohen M. B, and Dwyer M. B, "Evaluating improvements to a meta-heuristic search for constrained interaction testing", Empirical Software Engineering, Vol. 16, No. 1, pp. 61-102, 2011.

[10]. Bryce R. C, Sampath S, Pedersen J. B, and Manchester S, "Test suite prioritization by cost-based combinatorial interaction coverage", International Journal of System Assurance Engineering and Management, Vol. 2, No. 2, pp. 126-134, 2011.

[11]. Kuliamin V. V, and Petukhov A. A, "A survey of methods for constructing covering arrays", Programming and Computer Software, Vol. 37, No. 3, pp. 121-146, 2012.

[12]. Ahmed B. S, and Zamli K. Z, "A variable strength interaction test suites generation strategy using Particle Swarm Optimization", Journal of Systems and Software, Vol. 84, No. 12, pp. 217 -2185, 2011.

[13]. Yuan X, Cohen M. B, and Memon A. M, "GUI interaction testing: Incorporating event context", IEEE Transactions on Software Engineering, Vol. 37, No. 4, pp. 559-574, 2011.

[14]. Nie C, and Leung H, "A survey of combinatorial testing", ACM Computing Surveys (CSUR), Vol. 43, No. 2, pp. 11.1-11.29, 2011.

[15]. Torres-Jimenez J, and Rodriguez-Tello E, "New bounds for binary covering arrays using simulated annealing", Information Sciences, Vol. 185, No. 1, pp. 137-152, 2012.

[16]. Pachauri A, and Srivastava G, "Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism", Journal of Systems and Software, Vol. 86, No. 5, pp. 1191-1208, 2013.

[17]. Mao C, Yu X, Chen J, and Chen J, "Generating test data for structural testing based on ant colony optimization", In Proceedings of IEEE International Conference on Quality Software (QSIC), pp. 98-101, 2012.

[18]. Ahmed B. S, Zamli K. Z, and Lim C. P, "Application of Particle Swarm Optimization to uniform and variable strength covering array construction", Applied soft computing, Vol. 12, No. 4, pp. 1330-1347, 2012.

[19]. Baker R, and Habli I, "An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software", IEEE Transactions on in Software Engineering, Vol. 39, No. 6, pp. 787-805, 2013.

[20]. Fraser G, and Arcuri A, "Achieving scalable mutation-based generation of whole test suites", Empirical Software Engineering, Vol. 20, No. 3, pp. 783-812, 2015.

[21]. Debroy V, and Wong W. E, "Combining mutation and fault localization for automated program debugging", Journal of Systems and Software, Vol. 90, pp. 45-60, 2014.

[22]. Belli F, Budnik C. J, Hollmann A, Tuglular T, and Wong W. E, "Model-based mutation testing–approach and case studies", Science of Computer Programming, pp. 1-24, 2016.

[23]. Habibi E, and Mirian-Hosseinabadi S. H, "Event-driven web application testing based on model-based mutation testing", Information and Software Technology, Vol. 67, pp. 159-179, 2015.