Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

An Adaptive PSO-based Approach for Data Flow Coverage of a Program

Sapna Varshney[1] Monica Mehrotra[2] Charu Saini[3]
[1,2]Department of Computer Science, Jamia Millia Islamia, India
[3]Department of Teacher Training & Non-formal Education,
Jamia Millia Islamia, India

***Abstract* - Software testing is an important and expensive activity of the software development life cycle. Software testing includes test data generation and application of a test adequacy criterion. There has been an extensive application of meta-heuristic search algorithms to generate software test data for branch coverage and path coverage test adequacy criteria. However, test data generation for data-flow coverage test adequacy criterion remains a challenging task. Genetic algorithm and its variants have been the choice of researchers for automated test data generation. In recent years, other highly-adaptive swarm intelligence techniques such as Particle Swarm Optimization has also been applied for automated test data generation. In this paper, Particle Swarm Optimization algorithm with adaptive inertia weight strategy is used to generate test data for data-flow dependencies of a program. The proposed approach is evaluated on a set of benchmark programs, the measures considered are mean number of generations and mean percentage coverage achieved. The performance of the proposed approach is compared with that of Genetic Algorithm and random search. Over several experiments, it is shown that the proposed approach performed significantly better than random search and Genetic Algorithm in data-flow test data generation and optimization with an increasing performance gap for more complex subject programs.**

Keywords: Search Based Software Testing, Evolutionary Algorithms, Particle Swarm Optimization, Data-Flow coverage

I. INTRODUCTION

The aim of software testing is to identify all existing defects by checking the software for every possible input. However, exhaustive testing is expensive and impractical. The goal of software testing is to design an optimal test suite according to a test adequacy criterion [1]. There have been constant attempts to reduce the efforts and time required for software testing by automating the process of software test data generation; the efforts being constrained by the increasing size and complexity of the modern software.

Test data generation can be formulated as an optimization problem that is computationally hard [2]. Meta-heuristic search-based algorithms have been widely applied to generate optimal test suite for efficient software testing [3]. Most of the early test data generators employed Tabu Search, Hill Climbing and Simulated Annealing [4, 5] for test data generation. However, these algorithms could return a local optimal solution [6]. Therefore, evolutionary search-based algorithms, such as Genetic Algorithm (GA), have been applied as an effective alternative for test data generation and optimization [7 - 17]. Due to an extensive application of search-based algorithms to test data generation problem, the approach has come to be known as Search Based Software Testing (SBST, term coined by Harman and Jones in 2001) [18] and includes *Evolutionary Testing* as a sub-field. More recently, the focus is on the use of other highly adaptive search-based optimization techniques such as Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO). PSO is significantly simpler to implement with fewer numbers of parameters to adjust than GA. PSO has been shown to be well suited for test data generation with performance even better than GA [19 - 23].

Test data is generated according to a test adequacy criterion (encoded as a *fitness function* to be maximized or minimized) that is used to guide the search. Structural testing, so far, has been the main focus of SBST [3, 24, 25]. The adequacy criteria for structural testing can be *statement coverage* (every statement is executed at least once); or *branch coverage* (every possible outcome of all the predicates is exercised at least once); or *path coverage* (every possible path is traversed at least once). *Data-flow coverage* is a more effective and robust test adequacy criterion that focuses on the definition and usage of variables in a program [2, 26].

In this paper, an improved PSO-based approach with adaptive inertia weight strategy is proposed to generate test data for data-flow coverage of a program. The search for optimal test data is guided by a novel fitness function (proposed in our earlier work [17]) that is based on the concepts of dominance relations and branch distance. The performance of the proposed approach is compared with that of an elitist GA and random search. It is shown that the proposed adaptive PSO-based approach guided by the novel fitness function outperformed elitist GA and random search in terms of mean percentage coverage and mean number of generations of the algorithm to produce the final test suite for data-flow coverage of a program.

The rest of the paper is organized as follows: Section II summarizes related work. Section III provide a brief description of PSO with adaptive inertia weight strategy formulated for this study. Section IV describes data-flow analysis. Section V describes the proposed approach. Section VI gives the experimental results and discussion of experimental results. Section VII gives the conclusion.

II. RELATED WORK

In the context of structural testing, both static and dynamic methods of testing can be employed to generate test data for a program in accordance with a test adequacy criterion.

In *symbolic execution* [5], a static method, test data generation problem is formulated as a constraint satisfaction problem; however, the performance is constrained by many

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

programming constructs such as pointers, loop conditions that involve input variables, array subscripts, and procedure calls. Dynamic methods [9, 15] are based on program execution and can be classified as *random*, *path-oriented* and *goal-oriented* test data generation techniques. A *random test data generator* arbitrarily selects test data from the input domain. It is easy to implement but may fail to find optimal test data as the information about testing requirements is not incorporated into the test data generation process. A *path-oriented test data generator* [12] uses control flow information to identify a set of paths to be traversed and generates test data for these paths. However, a path-oriented test data generator will not work well for programs that have infeasible paths or paths that contain loops. A *goal-oriented test data generator* [9, 15 - 16] generates test cases that cover a selected goal such as a statement or a branch, irrespective of the path taken.

A search-based test data generation technique is guided by a *fitness function* that is designed in accordance with the chosen test adequacy criterion. From the literature, it can be inferred that branch coverage and path coverage are the most often used and well-understood measures for search-based structural test data generation [3]. There is a well-established standard way to calculate branch coverage based on approximation level and branch distance for a target branch from the program's CFG [2]. Data-flow coverage criterion, however, has not received the same attention [2, 24, 25] due to lack of publicly available tool support and difficulty in writing test cases that satisfy data-flow criteria. The measures that have been used to assess the cost of a search-based technique for test data generation problem are number of iterations, total number of fitness evaluations performed over all iterations to achieve the test adequacy criteria, search time and size of the optimal test suite [3]. Recently only there has been more work on search-based test data generation for data-flow coverage using GA as the algorithm of choice [8, 11, 13, 14, 17]. Now, PSO [23] and ACO [27] are also being applied and tested for test data generation due to faster convergence.

In an earlier work, Wegener et al. [10] defined different types of fitness functions for structural testing; data-flow test criteria being classified as node-node-oriented methods.

Windisch et al. [19] applied PSO to artificial and complex industrial test objects to generate test data for branch coverage. Their results showed efficiency and efficacy of PSO over GA for most code elements to be covered.

Agarwal et al. [20] applied binary PSO and, Agarwal and Srivastava [21] applied discrete quantum PSO to generate test data for branch coverage.

Mao [22] adopted PSO for test data generation with branch coverage as the test adequacy criterion.

Nayak and Mohapatra [23] proposed an algorithm to generate test cases using PSO for data flow testing. This technique cannot rank test cases because the fitness function, as taken from Girgis [13], assigns the same fitness value to all the test cases that cover the same number of def-use paths and a fitness value of 0 to all the test cases that do not cover any def-use path or cover a partial aim. Here, the fitness function is not able to guide the search.

Here, a PSO-based approach with adaptive inertia weight is proposed to generate test data for data-flow testing. The optimality of test data is measured in terms of mean percentage coverage and mean number of generations. The fitness function is designed using the dominance concepts augmented with branch distance which produces a smoother landscape for guiding the search [17]. This leads to faster and better convergence of test data to achieve the desired coverage. This is the main contribution of this paper.

III. PARTICLE SWARM OPTIMIZATION

In 1995, Kennedy and Eberhart [28] introduced the Particle Swarm Optimization algorithm based on the social and cognitive behavior of different swarms such as flock of birds, herd of animals or school of fishes. PSO was inspired by how the different members of a swarm co-operate with each other to achieve the optimal food locations. Unlike GA, PSO does not use evolution operators such as crossover and mutation. Instead, each member of the swarm (called particle) attains optimal solution by learning from its own experience and the experience of other members of the swarm. Each particle maintains its current position, current velocity and the best position it has achieved so far, called pbest. The global best position of the swarm is called gbest. Both pbest and gbest are used by the particle in determining its next best position in the swarm.

Let's say a swarm consists of n particles denoted as (p_1, p_2... p_n). Position of the i^{th} particle in the d-dimensional space is denoted as $X_i = (X_{i1}, X_{i2}\ldots X_{id})$. Velocity associated with the i^{th} particle in the d-dimensional space is denoted as $V_i = (V_{i1}, V_{i2}\ldots V_{id})$. In dimension d, the personal best position and the global best position of the i^{th} particle are denoted as $pbest_{id}$ and $gbest_{id}$ respectively. The velocity and position of the i^{th} particle in dimension d can be updated by (1) and (2) respectively as given below.

$$V_{id} = w.V_{id} + c_1r_1\,(pbest_{id} - X_{id}) + c_2r_2\,(gbest_{id} - X_{id}) \quad (1)$$

$$X_{id} = X_{id} + V_{id} \quad (2)$$

where, c_1 and c_2 are positive learning constants called cognitive and social scaling parameters chosen in such a way that their sum never exceeds 4, and r_1 and r_2 are two random numbers in the range [0,1]. The inertia weight w controls the impact of the previous history on the new velocity of the i^{th} particle. A particle's velocity in each dimension is clamped to a maximum magnitude V_{max}. The position and velocity of each particle in the swarm are continuously updated until an optimal solution is achieved.

Proposed Adaptive Inertia Weight Strategy

In PSO algorithm, a large value of inertia weight facilitates *exploration* (global search) of the input search space and a small value of inertia weight facilitates *exploitation* (local search) of the input search space for the optimal solution. Various inertia weighting strategies used in the literature have been categorized into *constant*, *random*, *time varying* and *adaptive* inertia weight strategies [29]. In adaptive inertia weight strategies, state of the particles in the search space (feedback mechanism) is used to adjust the value of the inertia weight.

In this study, an adaptive inertia weight strategy is adopted for faster convergence towards the optimal solution; fitness value of the particles is used to adjust the inertia weight. Ratio α of the particle's fitness to the average fitness of the swarm is calculated as shown in (3) below:

$$\alpha = f_i / f_{max} \quad (3)$$

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

Here, f_i = fitness of i^{th} particle and f_{max} is the maximum fitness achieved by the particles in the swarm. The range of α is [0, 1]. For lower values of α, increasing inertia weight can strengthen the particle's search capability. For values of α that are closer to 1, performance of the particles is better; so, a smaller inertia weight should be used. The inertia weight w_i for the i^{th} particle is therefore defined as a linear function of α and is calculated as follows:

$$w_i = 0.5\ (1-\alpha) + 0.5 \qquad (4)$$

The range of the inertia weight is [0.5, 1].

IV. DATA FLOW ANALYSIS

Data-flow analysis [26] augments the control-flow testing criteria; the emphasis is on the definition and use of variables in a program which could lead to more efficient and targeted test suites. However, the data-flow analysis is difficult and more expensive to perform. A variable is said to be defined in a program statement (*def-node*) if a value is associated with the variable. A variable is said to be used in a program statement if the value of the variable is referenced for computational use (c-use node) or a predicate use (p-use node). A *def-clear path* is a path from a definition occurrence to a use occurrence such that there is no other intermediate definition of the corresponding variable; notion of *killing definitions*. A def-clear path can be further categorized as a *dcu-path* (c-use of the variable) or a *dpu-path* (p-use of the variable). A dpu-path is formed for both the true and false branches of the predicate node. Data-flow testing make reference to the *control flow graph* (CFG) [26] of the program under test.

```
#include<stdio.h>
#include<conio.h>

1   void main() {
2     intn, product, j;
3   printf("\nEnter the number: ");
4   scanf("%d", &n);
5   product = 1;
6     if (n > 0) {
7       for (j=1; j<=n; j++) {
8         product = product * j;
9       printf ("\nFactorial = %d", product);
10    }
11    else
12      printf ("\nInvalid number.");
13  }
```

Fig. 1. Factorial program

Control Flow Graph (CFG):

Control flow graph is a directed graph G (V, E) that describes the flow of control through a program structure. Each node corresponds to one or more program statements that execute sequentially without any halt or internal branching except at the end. Each edge between two nodes represents the flow of control from one node to another. Predicate nodes correspond to the program statements that test a predicate (condition) and have more than one outgoing edge. A CFG has two unique nodes: an entry node n_0 and an exit node n_{end}, at which the program execution starts and halts respectively.

Dominator Tree:

For a directed graph G (V, E) with two unique nodes: the entry node n_0 and the exit node n_{end}, node n dominates node m (dominance relationship) if every path from entry node n_0 to m contains n. By applying dominance relationship to all the nodes of *G*, a tree can be obtained that is rooted at n_0. This tree is called the dominator tree denoted as *DT* (*G*) [30]. Dominator tree for a program is constructed using the CFG of the program. For each node m in the CFG:

- Dom (m): Set of all the nodes that dominate node m.
- For x,y ϵ Dom (m), either x ϵ Dom (y) or y ϵ Dom (x).

By definition, for any node m, m ϵ Dom (m).

Fig. 1 and Fig. 2 provide the example program and its instrumented version. Fig. 3 gives the CFG and Fig. 4 gives the dominator tree of the example program. Tables 1 and 2 provide the list of variables and all def-use paths respectively.

```
#include<stdio.h>
#include<conio.h>

1   void main() {
2     intn, product, j;
2   printf("\nEnter the number: ");
2   scanf("%d", &n);
2   product = 1;
3     if (n > 0) {
4       for (j=1; j<=n; j++) {
5         product = product * j;
6       printf ("\nFactorial = %d", product);
7   }
8   else
9     printf ("\nInvalid number.");
10  }
```

Fig. 2. Instrumented program

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

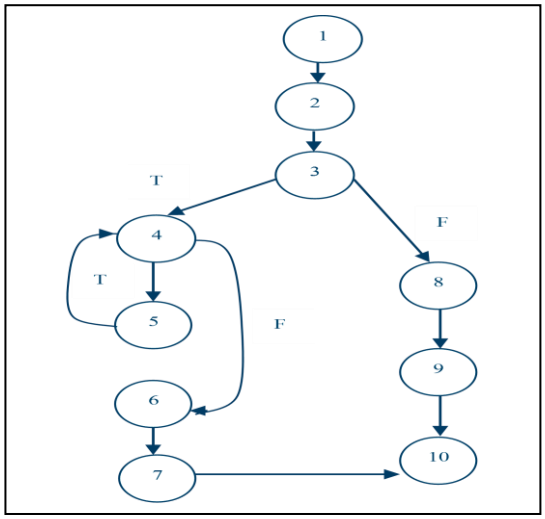


Fig. 3. CFG for the example program

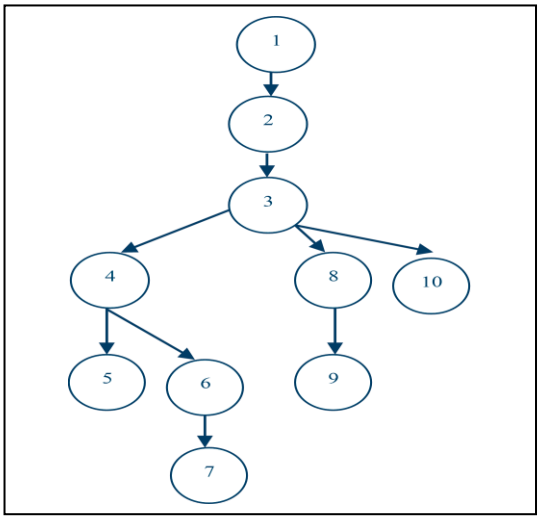


Fig. 4. Dominator tree for the example program

TABLE I. List of variables and def-use occurrences in the example program

Variable	def Node	c-use Node	p-use Edge
n	2	None	3-4 3-8 4-5 4-6
product	2, 5	5 6	None

TABLE II. All def-use paths for the example program

Path No.	def-use Path (Ends with -1 for c-use)	Killing Node(s)
1	2 3 4	None
2	2 3 8	None
3	2 4 5	None
4	2 4 6	None
5	2 5 -1	5
6	2 6 -1	None
7	5 6 -1	None

V. RESEARCH METHODOLOGY

An improved PSO-based approach with adaptive inertia weight is proposed to generate test data for data-flow dependencies of a program. The proposed approach is guided by a novel fitness function [18]. The algorithm accepts as input an instrumented program, dominator tree of the program, number of input variables, domain range of each input variable, list of def-use paths and killing nodes (if any) other than the algorithmic parameters. The output of the algorithm is a set of test cases (test-suite) for data-flow coverage of the program under test. The def-use paths are marked as covered or uncovered, if any.

An elitist GA-based test data generator guided by the same fitness function and random test data generator are also implemented for comparison to evaluate the efficiency and effectiveness of the proposed PSO-based approach with adaptive inertia weight strategy.

A. Design of the Proposed Fitness Function

The fitness function provides a measure of goodness of each candidate solution relative to the global optimal solution. According to Wegener et al. [10], def-use associations can be represented as node-node fitness functions. The distance to a node is represented by the standard fitness metric as given below:

nodeDistance=approach level+v (branch distance) (5)

This is a minimizing fitness function that evaluates to 0 if the target has been covered. Approach level gives the closest point (node with the critical branch) of a given execution to the target node. Whenever program execution follows a path which cannot lead to the target node, *branch distance* is calculated, measuring how close it came to traversing the alternate edge of the critical branch. The branch distance is commonly normalized in the range [0, 1] using a normalization function *v*, such that the approach level always dominates the branch distance. Table 3 [4] below shows the formulae used for computing branch distance for different predicates.

TABLE III. Branch distance measure for relational and logical predicates

S. No.	Predicate (C)	Branch Distance Formulae: f(C)
1	Boolean	if true then 0 else K
2	x = y	if (x-y)=0 then 0 else abs(x-y)+K
3	x ≠ y	if abs(x-y)≠0 then 0 else K
4	x > y	if (y–x)<0 then 0 else (y-x)+K
5	x ≥ y	If (y–x)≤0 then 0 else (y-x)+K
6	x < y	if (x–y)<0 then 0 else (x-y)+K
7	x ≤ y	if (x–y)≤0 then 0 else (x-y)+K
8	C1 && C2	f(C1) + f(C2)
9	C1 || C2	min(f(C1), f(C2))

* K is a failure constant that is added to branch distance if predicate is false

In this study, test data generation problem is taken up as a fitness maximization problem. The design of fitness function for data-flow dependencies of a program is based on the standard metric (5) and dominance relations between the nodes of the CFG for the program under test. The fitness function considers each def-use pair as two objectives. The first objective is to cover the dominance path of the *definition* node and the second objective is to cover the dominance path of the *c-use* node (dcu-path) or the dominance paths of the nodes of the *p-use edge* (dpu-path). For the selected def-use path, an *optimal test case (fitness value is 1)* covers all the nodes of the dominance paths of the two objectives; otherwise *branch*

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

distance is computed. If a *killing node* is traversed between the source node and the target node, a fitness value of 0 is assigned to the test case and it is discarded. Branch distance *bch(x, t_i)*, for test case t_i and target node x, is 1 if all the nodes of the dominance path of the target node are covered. Otherwise, branch distance is calculated at the critical branch C as the reciprocal of the value returned by an appropriate formula from Table 3 (for fitness maximization). Fitness functions are given by (6) and (7) below.

For dcu-path (d, u, v), where d is the *definition node* and u is the *c-use node* of a variable v, the fitness value ft (d, u, v, t_i) of test case t_i (i=1...p) is computed as follows:

$$ft(d, u, v, t_i) = \frac{1}{2} \times \left(\frac{|cdom(d, t_i)|}{|dom(d)|} \times bch(d, t_i) + \frac{|cdom(u, t_i)|}{|dom(u)|} \times bch(u, t_i) \right) \quad (6)$$

For dpu-path (d, (u_1, u_2), v), where d is the *definition node* and (u_1, u_2) is the *p-use edge* of a variable v, the fitness value ft (d, (u_1, u_2), v, t_i) of test case t_i (i=1...p) is computed as follows:

$$ft(d, (u_1, u_2), v, t_i) = \frac{1}{3} \times \left(\begin{array}{l} \frac{|cdom(d, t_i)|}{|dom(d)|} \times bch(d, t_i) + \frac{|cdom(u_1, t_i)|}{|dom(u_1)|} \\ \times bch(u_1, t_i) + \frac{|cdom(u_2, t_i)|}{|dom(u_2)|} \times bch(u_2, t_i) \end{array} \right) \quad (7)$$

dom(x) : dominance path of the target node x.
cdom(d, t_i) : nodes of dom(x) covered by test case t_i.
Branch distance *bch(x, t_i)* at critical branch C for test case t_i and target node x is the reciprocal of the value returned by an appropriate formula from Table 3.

B. Overall Algorithm

The proposed adaptive PSO-based algorithm accepts as input an instrumented program, dominator tree of the program, number of input variables, domain range of each input variable, list of def-use paths and killing nodes (if any) and the PSO parameters: population size, acceleration parameters and maximum velocity. Output is a set of test cases (test-suite) for data-flow coverage of the program under test. The def-use paths are marked as covered or uncovered, if any. A tool has also been developed for automated instrumentation of programs and generation of the program def-use paths. Dominator tree is generated manually. Infeasible paths, if any, are determined manually by careful analysis of the program.

The argument list *arg = (a_1, a_2... a_d)* is encoded into a d-dimensional position vector. An initial population of n particles and their velocities is randomly generated for the program under test P, where each particle and its velocity in the d-dimensional space is denoted as $X_i = [X_{i1}, X_{i2} \ldots X_{id}]$ and $V_i = [V_{i1}, V_{i2} \ldots V_{id}]$ for i=1, 2…n. For data-flow coverage criterion, design of fitness function is explained in the previous section. Initial value of *pbest* and *gbest* is 0. The adaptive inertia weight is calculated using (3) and (4). The algorithm is run once for each uncovered def-use path. If the selected path is not covered by any member of the current population, fitness value is computed for each member as explained earlier. Accordingly, for each particle in the position vector, the personal best position *pbest* and the global best position *gbest* can be updated. The evolution process continues until the selected def-use path is covered or maximum number of generations, whichever is achieved first. The other uncovered paths are also checked for coverage.

C. Study results

This section presents the experimental results on various subject programs. The measures collected are:

VI. EXPERIMENTAL SETUP

This section describes the subject programs, algorithmic parameters settings, experimental results and discussion. An elitist GA-based test data generator and random search are also implemented for comparison with the proposed approach.

A. Subject Programs

Details of the various programs used for this study are shown in Table 4 below. The programs are taken from the other researchers' work [7, 8, 12, 13, 22]. The programs have diverse features such as loops, equality conditions, logically connected and nested predicates.

TABLE IV. Subject programs

S. No.	Program	#Vars	#def-use Paths
1	Triangle Classifier	4	12
2	Quadratic Equation	5	20
3	Previous Date	5	66
4	Average Marks of 5 Subjects	7	19
5	Income Tax Calculator	8	34
6	Factorial	2	7

B. Parameters Tuning

Following settings have been used for PSO and GA in the proposed study after carrying out some preliminary experiments.

TABLE V. Algorithmic parameter settings

Algorithm	Parameters	Value
Common Parameters	Population Size	10, 15, 20, 25
	Max. number of generations	10^3
	Number of experiments for each program	100
PSO	Inertia weight	Given by (3) and (4)
	Acceleration constants: c_1, c_2	c_1=c_2=2.0
	Maximum velocity: V_{max}	Varies according to the program
GA	Parent selection strategy	Roulette Wheel
	Probability of crossover	0.8
	Probability of mutation	0.15
	Elitism (Individuals carried forward)	Upto 10% of population

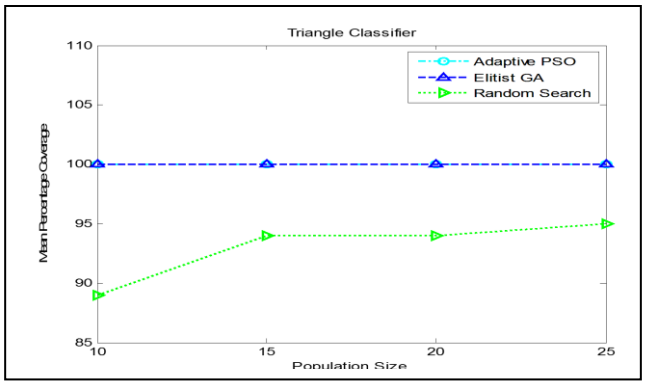
- *Mean percentage coverage:* Sum of the coverage achieved for each experiment over the number of experiments gives the mean number of generations for a particular subject program. A def-use path is marked as

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

covered the first time it is traversed and is not checked subsequently. For each subject program, infeasible uses, if any, are identified by careful manual analysis and are not considered while measuring data-flow coverage.

- *Mean number of generations:* The termination criterion for each experiment is either 100% data-flow coverage or 10^3 generations, whichever occurs first. Sum of the number of generations at termination for each experiment over the number of experiments gives the mean number of generations for a particular subject program. However, it does not indicate if the full data-flow coverage is achieved. Mean number of generations is higher for subject programs where full data-flow coverage is not achieved.

Detailed results for subject programs 'Triangle Classifier', 'Previous Date', 'Average Marks of 5 Subjects', and 'Income Tax Calculator' for different population sizes that are considered are presented in Figures 5 - 8 respectively. Table 6 summarizes the results of applying the proposed adaptive PSO-based approach, random search and GA for a population size of 25.



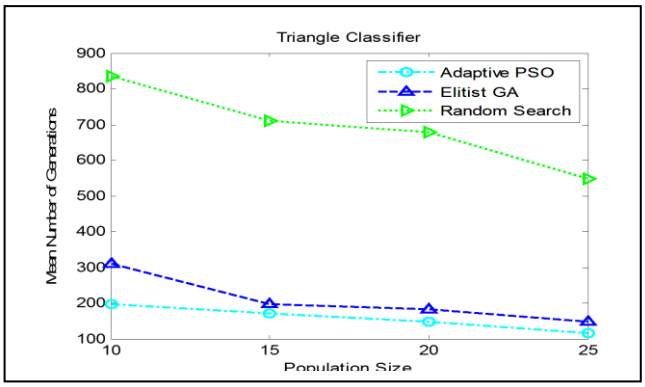
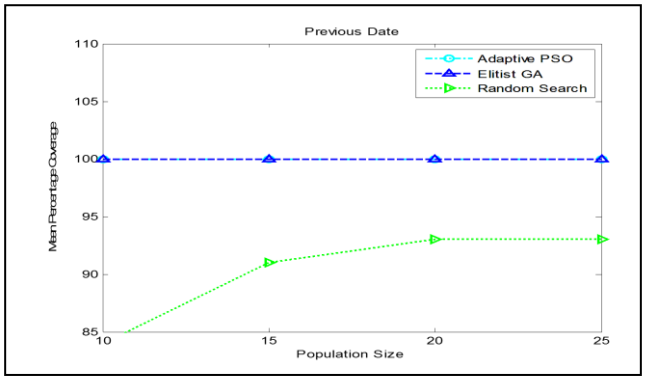


Fig. 5. Graphs for 'Triangle Classifier' program



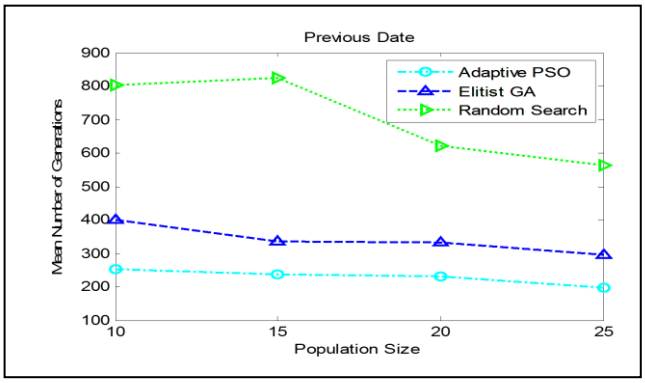
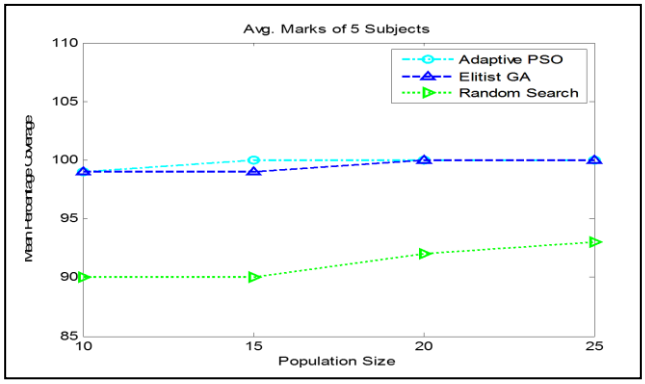


Fig. 6. Graphs for 'Previous Date' program



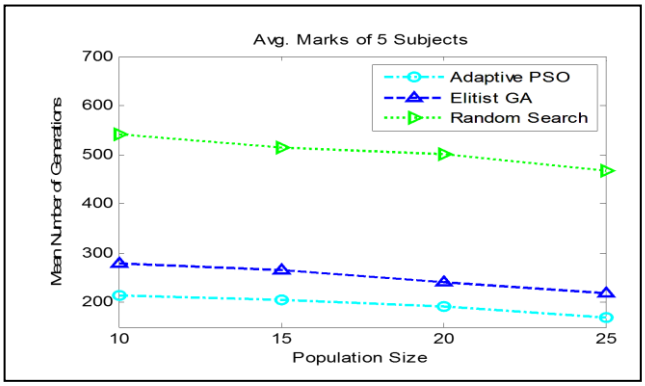


Fig. 7. Graphs for 'Average Marks of 5 Subjects' program

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

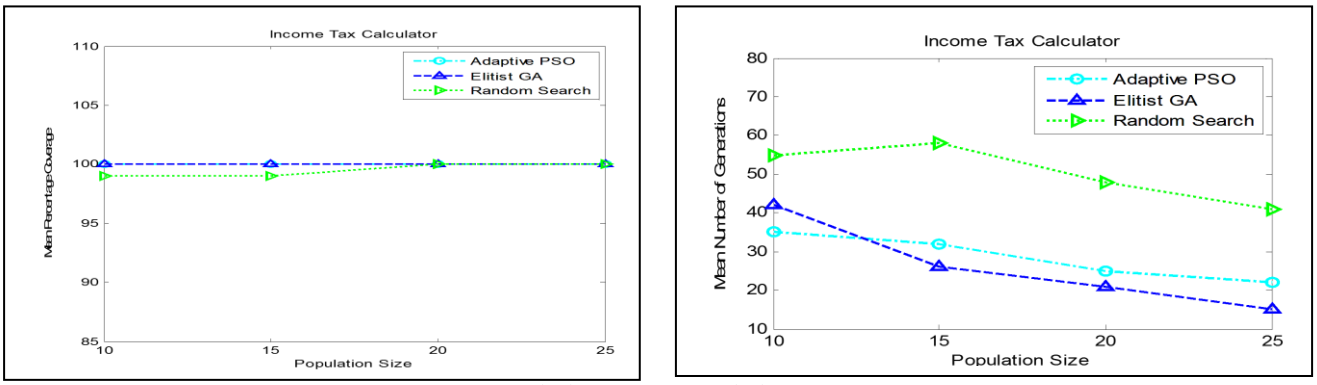


Fig. 8. Graphs for 'Income Tax Calculator' program

TABLE VI. Experimental results for various subject programs

S. No.	Program	#def-use Paths	Measure					
			Mean Number of Generations			Mean Percentage Coverage		
			Proposed Adaptive PSO	Elitist GA	Random Search	Proposed Adaptive PSO	Elitist GA	Random Search
1	Triangle Classifier	12	116	147	548	100%	100%	95%
2	Quadratic Equation	20	102	145	378	100%	100%	96%
3	Previous Date	66	198	294	563	100%	100%	93%
4	Average Marks of 5 Subjects	15	169	219	468	100%	100%	93%
5	Income Tax Calculator	34	22	15	41	100%	100%	100%
6	Factorial	7	4	5	17	100%	100%	100%

D. Discussion

Experimental results as presented above in Figures 5 - 8 and Table 6 are discussed in this section. With the novel fitness function, the proposed adaptive PSO-based approach and the elitist GA achieved full data-flow coverage for all the subject programs and for all population sizes that are considered. It can therefore be claimed that the novel fitness function is effective in achieving 100% data-flow coverage. For each program, infeasible uses, if any, were not considered while measuring data-flow coverage.

However, the mean number of generations to achieve full data-flow coverage is least with the proposed adaptive PSO-based approach for all the subject programs and for all population sizes that are considered (except for Program# 5 for population sizes 15, 20, 25). The proposed algorithm performs significantly better for programs with multiple and nested conditions such as 'Triangle Classifier', 'Quadratic Equation', 'Previous Date', and 'Average Marks of 5 Subjects'. As expected, the mean number of generations to achieve full data-flow coverage decreases as the population size increases due to a wider search space.

The performance of random search is worst with respect to mean number of generations to achieve same data-flow coverage for smaller population sizes and for programs with multiple and nested conditions. Random search did not achieve full data-flow coverage for smaller population size and for programs with multiple and nested conditions such as 'Triangle Classifier', 'Quadratic Equation', 'Previous Date', and 'Average Marks of 5 Subjects' programs. Random search achieved full data-flow coverage only for Programs 5 and 6 but with significantly higher number of generations.

VII. CONCLUSION

Research in the field of search based software testing for automated test data generation is relatively mature, but it is still an open problem. GA and its variations have been the focus of researchers to generate test data for control-flow coverage criteria. Data-flow coverage, however, has received relatively little attention. Recently only, other highly adaptive search-based optimization techniques such as PSO have been employed for structural test data generation. The main contribution of this paper is an improved PSO-based approach with adaptive inertia weight strategy guided by a novel fitness function to generate test data for data-flow coverage (all-uses criterion) of a program. The design of the novel fitness function is based on the dominance relations augmented with branch distance to provide a smooth search landscape.

The performance of the proposed PSO-based approach with adaptive inertia weight strategy has been experimentally evaluated and compared with that of random search and elitist GA for data-flow coverage. The experimental results have shown that the proposed adaptive PSO-based approach outperformed random search and elitist GA for data-flow coverage for all the subject programs in terms of the measured collected. In future, we intend to perform the study on real and more complex subject programs.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 6 Issue 02, February-2017

REFERENCES

[1] H. Zhu, A.V. Patrick, and H.R. Hall John, "Software unit test coverage adequacy", Computing Surveys, ACM, Vol. 29, No. 4, pp. 366-427, 1997.

[2] P. McMinn, "Search-Based Software Test Data Generation: A Survey", Journal of Software Testing, Verification and Reliability, Wiley, Vol. 14, No. 2, pp. 105-156, 2004.

[3] S. Ali, L. C. Briand, H. Hemmati, and R. K. P. Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation", IEEE Transactions on Software Engineering, Vol. 36, No. 6, pp. 742-762, 2010.

[4] N. Tracey, "A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software", Doctoral Thesis, University of York, 2010.

[5] R.A. DeMillo and A.J. Ofutt, "Constraint-based automatic test data generation", IEEE Transactions on Software Engineering, Vol. 17, No. 9, pp. 900-910, 1991.

[6] X.S. Yang, Engineering Optimization: An Introduction with Metaheuristic Applications, Wiley, New Jersey, 2010.

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

[8] A.S. Ghiduk, M.J. Harrold, and M.R. Girgis, "Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage", Proceedings of the 14th Asia-Pacific Software Engineering Conference, IEEE, pp. 41-48, 2007.

[9] B. Korel, "Automated Software Test Data Generation", IEEE Transactions on Software Engineering, Vol. 16, No. 8, pp. 870-879, 1990.

[10] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing. Information and Software Technology", Elsevier, Vol. 43, No. 14, pp. 841-854, 2001.

[11] K. Liaskos, M. Roper, and M. Wood, "Investigating Data-Flow Coverage of Classes Using Evolutionary Algorithms", Proceedings of the 9th annual conference on Genetic and Evolutionary Computation (GECCO'07), pp. 33-53, 2007.

[12] M.A. Ahmed and I. Hermadi, "GA-based multiple paths test data generator", Computers and Operations Research, Elsevier, Vol. 35, pp. 3107-3124, 2007.

[13] M.R Girgis, "Test Data Generation for Data Flow Testing Using a Genetic Algorithm" , Journal of Universal computer Science, Vol. 11, No. 6, pp. 898-915, 2005.

[14] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based Data-Flow Test Generation", International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp. 370-379, 2013.

[15] R.P. Pargas, M.J. Harrold, and R. Peck, "Test-Data Generation Using Genetic Algorithms", Journal of Software Testing, Verification and Reliability, Wiley, Vol. 9, No. 4, pp. 263-282, 1999.

[16] R. Ferguson and B. Korel, "The chaining approach for software test data generation", ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 1, pp. 63-86, 1996.

[17] S. Varshney and M. Mehrotra, "Search-based Test Data Generator for Data-Flow Dependencies using Dominance Concepts, Branch Distance and Elitism", Arabian Journal of Science and Engineering, Springer, Vol. 41, No. 3, pp. 853-881, 2016. DOI: 10.1007/s13369-015-1921-5

[18] M. Harman and B. F. Jones, "Search-based software engineering", Information & Software Technology, Vol. 43, No. 14, pp. 833–839, 2001.

[19] A. Windisch, S. Wappler, and J. Wegener, "Applying Particle Swarm Optimization to Software Testing", Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07), pp. 1121-1128, 2007.

[20] K. Agarwal, A. Pachauri, and Gursaran, "Towards Software Test Data Generation using Binary Particle Swarm Optimization", Proceedings of the XXXII National Systems Conference, pp. 339-343, 2008.

[21] K. Agarwal and G. Srivastava, "Towards Software Test Data Generation using Discrete Quantum Particle Swarm Optimization", Proceedings of the ISEC'10, pp. 65-68, 2010.

[22] C. Mao, "Generating Test Data for Software Structural Testing Based on Particle Swarm Optimization", Arabian Journal of Science and Engineering, Springer, Vol. 39, pp. 4593-4607, 2014.

[23] N. Nayak and D. P. Mohapatra, "Automatic Test Data Generation for Data Flow Testing Using Particle Swarm Optimization", Springer-Verlag Heidelberg, pp. 1-12, 2010.

[24] R. Malhotra and M. Khari, "Heuristic search-based approach for automated test data generation: a survey", International Journal of Bio-Inspired Computation, Inderscience, Vol. 5, No. 1, 2013.

[25] S. Varshney and M. Mehrotra, "Search Based Software Test Data Generation for Structural Testing: A Perspective", ACM SIGSOFT Software Engineering Notes, Vol. 38, No. 4, 2013.

[26] A. P. Mathur, Foundations of Software Testing, Pearson, 2008.

[27] A. Ghiduk, "A New Software Data-Flow Testing Approach via Ant Colony Algorithms", Universal Journal of Computer Science and Engineering Technology, Vol. 1, No. 1, pp. 64-72, 2010.

[28] J. Kennedy and R. C. Eberhart, "Particle Swam Optimization", Proceedings of IEEE International Conference on Neural Networks (ICNN'95), IEEE, pp. 1942-1948, 1995.

[29] A. Nickabadi, M. M. Ebadzadeh, and R. Safabakhsh, "A novel particle swarm optimization algorithm with adaptive inertia weight", Applied Soft Computing, Elsevier, Vol. 11, pp. 3658-3670, 2011.

[30] T. Lengauer and R.E. Tarjan, "A fast algorithm for finding dominators in a flowgraph", ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, pp. 121-141, 1979.