Application of Evolutionary Particle Swarm Optimization Algorithm in Test Suite Prioritization

Chug Anuradha(✉) and Narula Neha

University School of Information Technology, GGSIPU, Delhi, India
anuradha@ipu.ac.in, nehanarula665@gmail.com

Abstract. Regression testing is a software verification activity carried out when the software is modified during maintenance phase. To ensure the correctness of the updated software it is suggested to execute the entire test suite again but this would demand large amount of resources. Hence, there is a need to prioritize and execute the test cases in such a way that changed software is tested with maximum coverage of code in minimum time. In this work, Particle Swarm Optimization (PSO) algorithm is used to prioritize test cases based on three benchmark functions Sphere, Rastrigin and Griewank. The result suggests that the test suites are prioritized in least time when Griewank is used as benchmark function to calculate the fitness. This approach approximately saves 80% of the testing efforts in terms of time and manpower since only 1/5 of the prioritized test cases from the entire test suite need to be executed.

Keywords: Particle swarm optimization · Benchmark functions
Prioritization · Regression testing

1 Introduction

Software testing is the one of the most crucial phase in Software Development Life Cycle (SDLC). For the success of any software it is very important that the software is tested thoroughly with intent of finding maximum number of defects while testing. Software testing consumes approximately 50% of the software development efforts [1]. Changing business requirements and market trends urge the need for the software to undergo changes even after the software is delivered and becomes operational. Regression testing ensures to verify the correctness of the changed software and its corresponding affected parts after it becomes operational. Testing the whole software again demands a huge set of resources which is a big constraint in terms of time and money.

Test case prioritization (TCP) ensures that the test cases are arranged using priority value so that the test cases with highest priority are executed first. The test cases can be prioritized either randomly or based on the branch or statement coverage [2] of the software. Prioritizing test cases ensures that the time required to reach a performance goal is optimized [3]. The potential benefit of prioritizing test cases is that it ensures to rank the test cases rather than reducing or discarding the test cases from the test suite [4].

D. J. Hemanth and S. Smys (eds.), *Computational Vision and Bio Inspired Computing*,
Lecture Notes in Computational Vision and Biomechanics 28,
https://doi.org/10.1007/978-3-319-71767-8_2

Particle Swarm Optimization (PSO) is a stochastic population based optimization technique given by Kennedy and Eberhart in 1995 [5] that is motivated by social behavior of fish schooling and bird flocking. The population in PSO called as particles that fly through the problem domain or the search space to reach an optimum solution. PSO is widely used because of its implementation simplicity, dimensions scalability and performance in terms of empirical solutions for global search problems [6]. It is used to solve a number of complex problems like permutation-flop shop sequencing problem [5], real-time embedded systems [7], capacitor-problem [8], soft-sensor [9] and generating test data in the basis of data flow coverage [10].

In this paper, we have implemented PSO with three different Benchmark Functions (BFs) for calculating the fitness value of the test cases. BFs are used to find optimal solutions for optimization problems and to compare and analyze the effectiveness of different optimization algorithms. In this work, we have used three single-objective BFs that are Sphere [5, 11], Rastrigin [5] and Griewank [5, 11, 12]. In current work, for a set of fifteen JAVA programs we first calculate the number of independent paths by drawing control-flow graph (CFG) and decision to decision (DD) graph. For every independent path a set of fifty test cases are generated that are further prioritized using PSO. The test results are driven based on the time taken and the global best value obtained while prioritizing test cases with each BF. The results proof Griewank works best with PSO as it takes approximately 8.90% of the average time taken by Sphere function to prioritize the test cases. It is also proved that Griewank takes approximately 64.17% of the average time taken by Rastrigin function for prioritization.

The paper proposes a way to minimize the testing efforts invariably by substituting different fitness function for calculating fitness of the test cases based on their position. This work suggests of using Griewank as BF with PSO to prioritize test cases as it would help in saving a lot of time, effort and cost during critical project deliveries when resources availability is a crunch. In this work, the test cases are prioritized and not reduced or removed from the test suite.

The rest of the paper is organized as follow: in Sect. 2 literature reviews is discussed. Section 3 covers the concept of test suite prioritization using branch and statement coverage. Section 4 explains PSO algorithm and BFs in detail. Section 5 covers a case study on TCP. In Sect. 6 results are derived based on the case study. In Sect. 7 different threats to validity are discussed. In the last section paper concludes and suggests future work.

2 Related Work

An extensive research work is carried out on test suite prioritization, selection and minimization using different nature inspired meta-heuristic algorithms [13]. An extensive work has been done in the field of test case prioritization, selection and test data generation using PSO. In 2014, Mor [4] has evaluated the effectiveness of different test suite prioritization techniques using Average Percentage of Fault Detected (APFD) metrics. Walcott in year 2006 [14], has given an algorithm for time constraint aware prioritization. Sharma [15] has modified the algorithm for prioritizing the test cases based on timing and APFD metrics in 2014. Hassan et al. worked on Genetic

Algorithm (GA) and PSO and proved that PSO works best amongst the two in year 2006 [12]. Elbeltagia et al. [6] worked on GA, Memetic Algorithm (MA), Ant Colony Optimization (ACO) and PSO and proved that PSO works the best amongst the four algorithms. Nayak [16] has used PSO for automatic test data generation for data flow testing in year 2010. In 2014; Chawla [17] has given a hybrid algorithm for test data using soft-computing technique with PSO and GA.

Harman [18] has done a survey work in his paper in the area of regression testing minimization, prioritization and selection and suggested potential areas and scope for future work in them. In 2011, Kaur [19] has blended PSO with cross-over operator to avoid convergence of population to local best. In May, 2011 Arora [20] has given Hybrid Particle Swarm Optimization (HPSO) that is based on combination of PSO and GA to increase the search space. Routing problems, job-scheduling, task-scheduling problem [21] have been solved using PSO by many researchers in a distributed environment to solve tasks in efficient and cost-effective manner. El-Sherbiny [5] has given an algorithm for particle swarm optimization without using velocity for calculating position of particle. In 2004, Yang [22] has been used PSO to solve NP-hard problems like knapsack. In 2017, Kumar [23] has used PSO to solve NP-complete problem like test suite generation. In year 2007, Hendtlass [24] has worked on PSO algorithm with focus on counting total number of evaluations for calculating fitness.

As far as the literature survey is concerned El-Sherbiny [5] in his work has given an algorithm inspired by PSO for optimization problems that work without calculating velocity at which particle moves in the search space. He has efficiently reduced the number of iterations for calculating the best solutions for an optimization problem. In 2014, Mor [4] has used APFD metrics for measuring the rate of fault detection while prioritizing the test cases based on the coverage criteria provided by different prioritization techniques. He has concluded that higher values of APFD metric provide a better rate of fault detection. In 2006, Walcott [14] has given an algorithm for prioritizing test cases using GA in a time constraint environment for systems like PlanetLab and MonetDB that uses the concept of nightly builds and unit testing of the software. In 2010, Nayak [16] has simulated GA and PSO to generate automatic test data for data flow testing. His work proofs that PSO outperforms GA by 100% in def-use coverage. In 2014, Chawla [17] has proposed a hybrid algorithm based on PSO and GA to automate test data generation and the effectiveness of the algorithm is confirmed using percentage of fault coverage against unit of time and percentage of fault detected by generated test cases.

In 2011, Kaur [19] has given a Hybrid Particle Swarm Optimization Algorithm (HPSO) that combines the techniques for PSO and GA for widening the search space. In HPSO the initial population of PSO is mutated before performing rest of the steps of PSO algorithm to improve average percentage of fault detected. In 2010, Harman has performed both detailed analysis and survey in the field of regression test prioritization, selection and minimization. And his work suggested how the three terms are related to each other in terms of implementation. In 2012, Ming [25] et al. has combined mutative scale chaos and PSO to mitigate the problem of slow convergence and local optimum points of PSO algorithm. In 2007, Hendtlass [24] has used three fitness functions Sphere, Rastrigin and Schwefel's function in different number of dimensions. He has proved that best fitness evaluation is found using Schwefel's function. In the paper, we

have referred the research work done after the year 2000 on test data generation, prioritization and optimization during software testing.

3 Test Case Prioritization

To make regression testing cost-effective we need to prioritize the test cases in such a manner that after executing the prioritized test cases it's ensured that the changed part of the software is tested effectively. Test case coverage is a way to describe how well the code is covered by a given test suite. Therefore, while testing the software it becomes very essential to design an effective test suite that uncovers maximum number of defects in the software. To save the testing effort there is a need of ranking the test cases based on criteria that fulfills the prerequisite of uncovering maximum defects in the updated software. Different prioritization techniques are used for prioritizing test suite but in this work we focus on two aspects of test case coverage [7] as described below.

3.1 Prioritizing Based on Statement Coverage

Software coverage (SC) is based on the concept of executing all the statements in the code at least once for a given set of requirement. SC can be achieved with a basic knowledge of code structure and by using flowchart for the software workflow. SC is represented by a metrics that defines the number of statements covered by the test case in a code block. In any program SC is calculated using the formula specified in Eq. (1). For a given procedure P as shown in Fig. 1a the number of statements covered by a set of three test cases is represented using statement coverage metrics in Fig. 1b.

$$\text{Statement Coverage} = \frac{\text{Number of executed statement}}{\text{Total number of statements}} \times 100 \tag{1}$$

```
Procedure P
1 int a=5, b=0;
2 while (a>0)
3   If(a%2==0)
4         Print "Even"
5   Else
6         Print "Odd"
7   End If
8 End While
9 Print "Completed."
```

1(a) Procedure P

Statement Coverage			
	TC 01	TC 02	TC 03
1	X	X	X
2	X	X	X
3		X	
4		X	
5	X		
6	X		
7	X	X	
8	X	X	
9	X	X	X

1(b) Statement Coverage for Procedure P

Branch Coverage			
	TC 01	TC 02	TC 03
Entry	X	X	X
2 True	X	X	
2 False			X
3 True	X		
3 False		X	

1(c) Branch Coverage for Procedure P

Fig. 1 For a program P shown in **a** that consists of two branch nodes the corresponding matrix for calculating the statement and branch coverage of the program is depicted in **b** and **c** respectively

3.2 Prioritizing Based on Branch Coverage

Branch coverage (BC) is also known as all-edges coverage or decision coverage. BC focuses on the aspect of covering all the branches or simply all the true and false conditions in the code. Prioritizing test cases on the basis of BC focuses on ranking the test cases on the basis of total number of decision or branch nodes covered by the test case. In this study we have designed the test cases in such a manner that all the branch conditions in a program are covered by the test cases. A test suite that covers all the decision nodes in the software would automatically give 100% of SC. The BC for any given software can be calculated using the formula specified in Eq. (2). For the procedure P in Fig. 1a the BC is represented by listing all the branches and the corresponding test cases that cover these branched is shown in Fig. 1c.

$$\text{Branch Coverage} = \frac{\text{Number of decisions outcome exercised}}{\text{Total number of decision outcomes}} \times 100 \tag{2}$$

4 Particle Swarm Optimization

PSO is an evolutionary process derived on the basis of social behavior of birds migrating to a destination that is currently unknown [1]. Each bird in the swarm is called a particle and for each particle we need to optimize its position in the swarm. In PSO every particle in the swarm has a position and velocity associated with it in an n-dimensional space. Every particle in the swarm strives for two best values, the first one called as pBest that is the best fitness value retrieved by particle so far and another is called gBest that is the best fitness value achieved so far amongst all the particles.

In PSO the size of the swarm population is denoted as S, $S \in R$ (S belongs to set of real number) and the position and velocity vectors are defined as follow: $X_i = (x_{i1}, x_{i2}, x_{i3} \ldots x_{ik})$ and $V_i = (v_{i1}, v_{i2}, v_{i3} \ldots v_{ik})$. On the basis of position of the particle in the swarm the fitness is calculated using fitness function and is denoted as $F_i = (f_{i1}, f_{i2}, f_{i3} \ldots f_{ik})$. The fittest particle found at a given point of time is denoted as $F_g = (f_{g1}, f_{g2}, f_{g3} \ldots f_{gk})$.

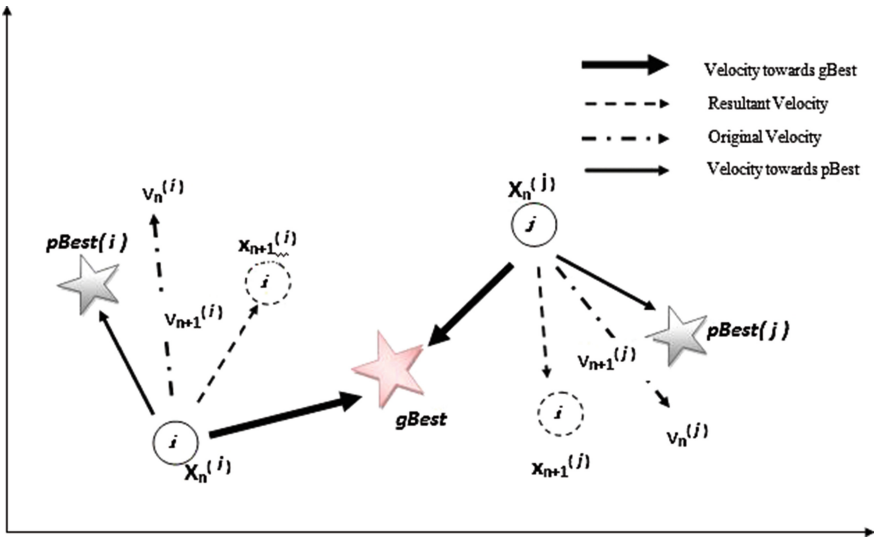

Fig. 2 The typical movement of two particles $X_n^{(i)}$ and $X_n^{(j)}$ particles in a swarm is depicted that changes its position and velocity to reach the global best value

The typical scenario in PSO is represented in Fig. 2 for two particles i and j that fly in the search space to reach a gBest. To understand the movement of a particle in swarm towards gBest lets understand how particle i move in search space. The particle *i* have an initial position and velocity denoted by X_n and V_n. It is moving towards the gBest by updating its position to X_{n+1} and the resultant velocity to V_{n+1}. The best position of the particle i attained so far is maintained in variable pBest.

A pseudo code for a generic PSO algorithm is depicted as follow. The algorithm begins by initializing the swarm population with velocity, position and fitness value. After the population is initialized the particle's velocity and position is updated for a number of iterations. The algorithm checks and updates the pBest of particle if it is greater than its current fitness value. Similarly if pBest is less than gBest the algorithm updates gBest value with pBest value of current particle.

Generic PSO Algorithm:
Input: A swarm of particles that fly in search space to reach a position that is currently unknown.
Output: The global best position in the swarm is found.
Begin
1. For i = Total number of particles
2. For j=1 to Total dimensions
3. Initialize the velocity, position and fitness of the particle.
4. Initialize the particles personal best cost and personal best position same as initial position and velocity.
5. End, End
6. Iteration = 0
7. While (iteration < Maximum Number of Iterations)
8. For i = 1 to Total number of particles
9. For j=1 to total dimensions
10. Update particle's position and velocity.
11. Calculate particle's fitness value
12. If (Fitness (*particle*) < Fitness (*pBest*))
13. Update pBest with current position of particle.
14. If (Fitness (*pBest*) < Fitness (*gBest*))
15. Update global best position i.e., gBest with pBest value.
16. End, End, End, End
17. Until iteration= Maximum Number of Iterations

4.1 Mapping PSO to Test Suite Prioritization

In this work, we have used PSO for prioritizing the test cases to save testing efforts during regression testing. To understand how PSO fits in Test Suite Prioritization (TSP) problem; in this section mapping of PSO to TSP is explained. In the test suite the test cases are equivalent to particles in the swarm. For every test case we calculate initial velocity and position using formula's defined in Eqs. (3) and (4) respectively and the initial velocity of the test case also incorporates the average value of the test data. The aim is to reduce the cost associated with the test case that is equivalent to the fitness value in generic PSO algorithm specified in pseudo code Generic PSO Algorithm. The initial fitness value for the test case is calculated using formula specified in Eq. (5) that takes position of the test case and dimension as an input.

$$v_{ik} = (\text{maxX} - \text{minX}) * \text{rand} + \text{minX} + (\text{mean}) \tag{3}$$

$$x_{ik} = (\text{maxX} - \text{minX}) * \text{rand} + \text{minX} + (\text{mean}) + v_{ik} \tag{4}$$

$$f_{ik} = \text{CostFunction}(x_{ik}, \text{dimension}) \tag{5}$$

The process to find candidate solution is repeated for a set of iterations and during each iteration particle updates its velocity and position on the basis of formula given in Eqs. (6) and (7) [7] respectively.

$$V_{ik} + 1 = w.V_{ik} + c_1.\text{rand}.(f_{ik} - x_{ik}) + c_2.\text{rand}.\left(f_{gk} - x_{ik}\right) + (\text{mean}) \tag{6}$$

where

V_{ik}	velocity of particle i at iteration
w	weighing function
c_1, c_2	weighing factors
f_{ik}	particles current best position
x_{ik}	particles current position
f_{gk}	global best
rand	random number in range [0, 1]

$$X_{ik+1} = x_{ik} + v_{ik+1} \tag{7}$$

The value of weighing function, w is calculated using formula given in Eq. (8). The weighing factor suggest how well the particle is moving towards the best solution.

$$w = \text{minX} - (t/\text{MaxGeneration}) * (\text{maxX} - \text{minX}) \tag{8}$$

where

minx	Initial weight
maxX	Final weight
MaxGeneration	Total number of iterations
t	represents current iteration.

The parameters and variables that are used in implementing PSO for test suite prioritization are listed in Table 1.

Table 1 PSO parameters

Weighing factors	c_1 = 2.0, c_2 = 2.0	Dimension	1
Initial weight (minX)	0.0	No of particles/test cases	50
Final weight (maxX)	2.0	Number of Iterations	20

4.2 Fitness Calculation in PSO

The fitness or the cost value of a particle in PSO can be calculated using different single or multi-objective functions. The objective function is also known as benchmark

function and is used for evaluating the effectiveness of optimization problems based on factors such as performance, convergence rate, precision and robustness.

In this work, the BFs are used to calculate the cost value associated with the test case based on the position and velocity of test case in the test suite. In this paper, we have used three single-objective functions as shown in Table 2 [11, 24] to calculate the fitness of the test cases using formula specified in Eq. (5). The performance of the three functions is compared with each other in terms of time to identify which work best with PSO for prioritizing the test cases.

In this work, the BFs are used to calculate the cost value associated with the test case based on the position and velocity of test case in the test suite. In this paper, we have used three single-objective functions as shown in Table 2 [11, 24] to calculate the fitness of the test cases using formula specified in Eq. (6). The performance of the three functions is compared with each other in terms of time to identify which work best with PSO for prioritizing the test cases.

Table 2 Benchmark functions

Function	Characteristics	Dim	Range
Sphere $f(\mathrm{x}) = \sum_{i=1}^{d} x_i^2$	Continuous, convex, unimodal	1	$[-100, 100]^{\mathrm{D}}$
Rastrigin $f(\mathrm{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right]$	Local minima, highly multimodal	1	$[-5.12, 5.12]^{\mathrm{D}}$
Griewank $f(\mathrm{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	Continuous, differentiable, non-separable, scalable, multimodal, regularly distributed local minima's	1	$[-600, 600]^{\mathrm{D}}$
Optimal Value for all functions is 0			

4.3 Proposed Algorithm

The proposed algorithm for prioritizing test cases using PSO is given in pseudo code as follow [25, 26]. The process of prioritizing test cases begins by identifying independent paths in program by drawing CFG and DD-graph. For the identified paths the test cases are generated based on decision nodes in the code that guarantees 100% BC and SC [7].

Proposed PSO Algorithm:
Input: A number of test suites based on number of independent paths in the program.
Output: Test cases prioritized on the basis of cost or fitness value.
Begin

1. Identify independent paths of the program under test.
2. For every independent path generate test data based on branch condition in the program.
3. For paths= Number of independent paths
4. For i = Total number of test cases
5. For j=1 to Total dimensions
6. Initialize the velocity and position for each test case and calculate the cost associated with test case using a benchmark function specified in table 2.
7. Initialize the test case personal best cost and personal best position same as initial cost and position.
8. End
9. End
10. Iteration = 0
11. While (iteration < Maximum Number of Iterations)
12. For i = Total number of test cases
13. For j=1 = Total dimensions
14. Pick random number in range [0,1] and update position and velocity of the test case.
15. Update cost value of the test case using updated particle's position for the selected benchmark function at step 3.
16. If (Fitness (*particle*) < Fitness (*pBest*))
17. Update test case best known position i.e., pBest with current position of test case.
18. If (Fitness (*pBest*) <Fitness (*gBest*))
19. Update global best position i.e., gBest with pBest of current test case.
20. End
21. End
22. End
23. End
24. Until iteration= Maximum Number of Iterations
25. End //Repeat steps 4 -24 for every path in program.

The algorithm begins by initializing velocity, position and fitness using formula 3, 4, and 5 for each test case on the path. The initial value for pBest position and pBest cost of the test case is same as that of the test case position and cost. At step 11, the counter iterates for a pre-defined number of iteration. For every iteration the velocity and position of the test cases are updated using formula 6 and 7. The algorithm verifies if pBest fitness value of the test case is greater than current fitness value of test case; the pBest is updated with current fitness of test case. Similarly, the algorithm validates for fitness value of gBest and pBest; and updates gBest with pBest fitness of the test case. The algorithm also incorporates the dimension variable while iterating the test cases.

4.4 Flow Chart of Proposed Algorithm

The flow chart for the algorithm proposed in this work is shown in Fig. 3. The algorithm begins by initializing the test cases or swarm population with random values. The global best is assigned a maximum value and aim is to minimize the value for gBest after performing a set of iterations. The result achieved at the end of the algorithm is a set of prioritized test cases sorted on the basis of increasing order of pBest.

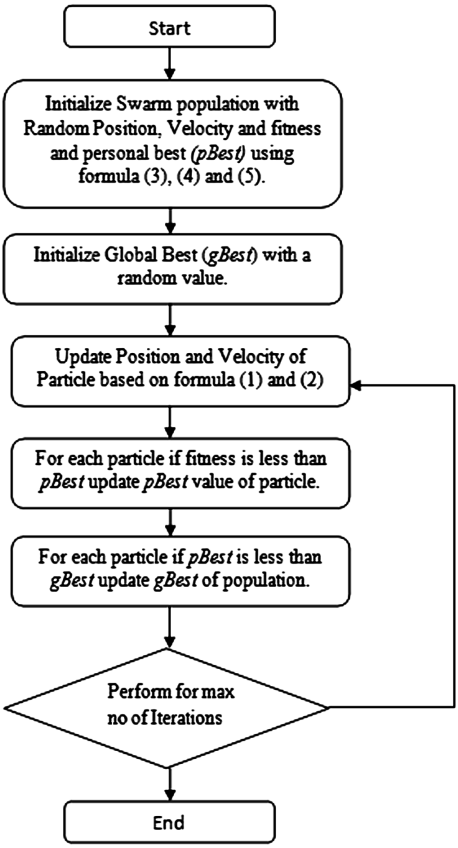

Fig. 3 Flow chart for PSO prioritization

5 Analysis and Evaluation

This section includes the steps of data collection followed by implementing PSO for a case study to derive results and showcase how the test cases are prioritized on the basis of fitness value using different BFs.

5.1 Data Collection

In this work, we have implemented PSO for a set of fifteen JAVA programs using Eclipse as an Integrated Development Environment (IDE). The test programs along with their number of input and their independent paths are listed in Table 3. The PSO algorithm is performed for twenty iterations for every test path and the dimension value used is 1.

Table 3 Test programs

Program no	Program	No of inputs	No of paths
P1	A calculator program	3	6
P2	Program to find leap year or not	1	4
P3	Calculate area and circumference of circle	3	4
P4	To check number is a Armstrong no	1	3
P5	To calculate factorial of number	1	2
P6	To swap two numbers	2	2
P7	To generate Floyd triangle	1	2
P8	Student classification problem	3	6
P9	To compare two numbers	2	3
P10	To generate fibonacci series	1	2
P11	To generate inverted triangle	3	3
P12	To check whether a number is palindrome or not	1	3
P13	Roots of a quadratic equation	3	3
P14	Triangle classification problem	3	5
P15	Find square root of number	1	2

5.2 Case Study: Calculator

In this work, calculator program is used as a case study for understanding PSO implementation and the code for which is shown in Fig. 4. The program has 32 Line of Code (LOC) and expects three inputs from the user; input 1 specifies the operation and input 2 and input 3 are integers on which operation needs to be performed. The DD-graph is drawn as shown in Fig. 5 that serves the basis of generating the test cases based on branch nodes.

Calculator.java

```
1. public class Calculator {
2. public static void main(String args[]) {
3. Scanner in = new Scanner(System.in);
4. int num1 = 0, num2 = 0, operation = 0;
5. double result = 0.0;
6. System.out.println("Please enter a choice value:" + "\n" + "1.Add" +
"\n" + "2.Subtract" + "\n" +      "3.Multiply"+ "\n" + "4.Division");
7. operation = in.nextInt();
8. if (operation == 1 || operation == 2 || operation == 3 || operation ==
4)
9. {        System.out.println("Enter 1st Number in range of 1-100");
10.         num1 = in.nextInt();
11.         System.out.println("Enter 2nd number in range of 1-100");
12.         num2 = in. .nextInt();
13.         if (num1 < 1 || num1 > 100 || num2 < 1 || num2 > 100)
14.         System.out.println("Invalid input values entered");
15.         else {
16.         switch (operation)
17.                 {
18.                 case 1:
19.                         result = (double) num1 + num2; break;
20.                 case 2:
21.                         result = (double) num1 - num2;break;
22.                 case 3:
23                          result = (double) num1 * num2;break;
24.                 case 4:
25.         result = (double) num1 / num2; break;
26. }
27.                 System.out.println("Result : " + result); }
28.}
29. else
30.         System.out.println("Invalid operation selected");
31.}
32.}
```

Fig. 4 Calculator program implemented in JAVA

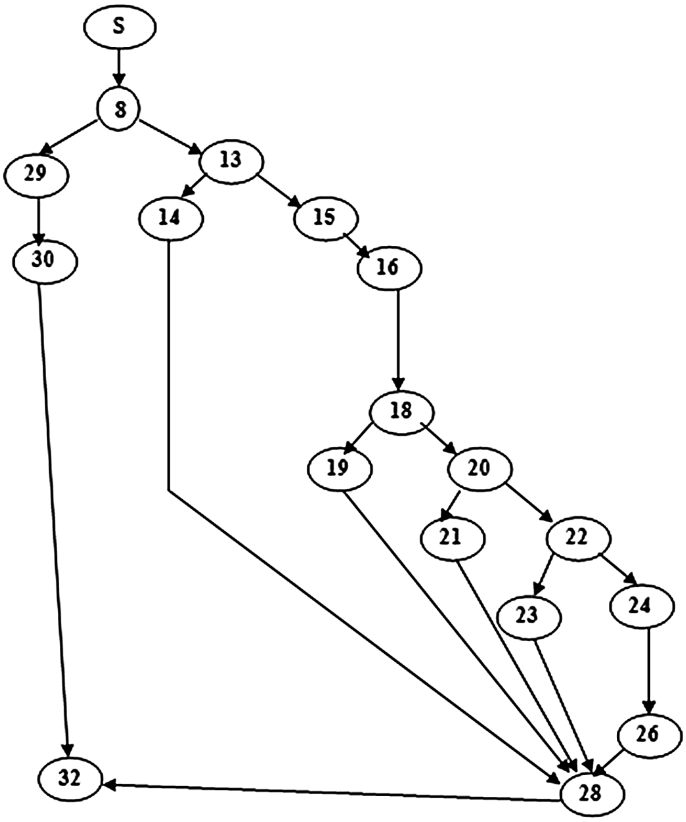

Fig. 5 DD-graph for program calculator.java

In Table 4 the independent paths for the DD-graph drawn in Fig. 5 are shown. For every identified path in Table 4 a set of 50 test cases are generated using decision nodes as boundary criteria while generating test cases.

Table 4 Independent paths for calculator.java

IP1	S 8 29 30 32	IP4	S 8 13 15 16 18 20 21 28 32
IP2	S 8 13 14 28 32	IP5	S 8 13 15 16 18 20 22 23 28 32
IP3	S 8 13 1516 18 19 28 32	IP6	S 8 13 15 16 18 20 22 24 25 26 28 32

In Table 5 two test cases for each path based on the boundary condition is shown along with the average value for the test data is given. The mean value is valued for initializing the velocity of the test case using Eq. (3). The values for input 2 and input 3 are in range of [0, 100].

Table 5 Test data for the independent paths

Path	Boundary condition	Test case	Parameters			Mean (Input to PSO)
			Input 1	Input 2	Input 3	
IP1	Input 1 <> {1, 2, 3,4}	1	5	7	8	6.66
		2	8	8	9	8.33
IP2	Input 1 = {1, 2, 3, 4}; Input 2 > 100 || input 2 < 0; input 3 > 100 || input 3 < 0	1	1	−25	25	−16.33
		2	3	24	101	42.67
IP3	Input 1 = 1; 0 <= input 2 <= 100; 0 <= input 3 <= 100	1	1	1	2	1.33
		2	1	3	4	2.67
IP4	Input 1 = 2; 0 <= input 2 < = 100; 0 <= input 3 < = 100	1	2	1	2	1.67
		2	2	3	4	3
IP5	Input 1 = 3; 0 <= input 2 <= 100; 0 <= input 3 <= 100	1	3	2	1	2
		2	3	4	3	3.33
IP6	Input 1 = 4; 0 <= input 2 <= 100; 0 <= input 3 <= 100	1	4	2	1	2.33
		2	4	4	3	3.66

5.3 Empirical Results

The PSO algorithm specified in Fig. 4 is run on all the independent paths of Table 4. The algorithm takes mean value of the test data generated for calculating the velocity, position and fitness of the test cases. At the end of the algorithm we get test eighteen prioritized test suite that includes three test suites for each path based on three BFs

applied on a test suite. The results drawn are not repeatable since calculations for PSO variables uses random numbers. The time taken for prioritization may vary depending on the test environment and machine configuration.

In Table 6 the prioritized test cases for six paths in the calculator program using PSO for three BFs is shown. The results show top ten prioritized test cases amongst the fifty test cases for each path. The gBest value achieved for all the paths using the BFs is listed in the table and the gBest for each path matches the pBest value of the first prioritized test case. The total time taken for prioritizing all the test paths using Sphere, Rastrigin and Griewank is summarized in the last row of the Table.

Table 6 Test suite prioritized for calculator program using PSO

Path	Sphere		Rastrigin		Griewank	
	Prioritized test cases	gBest	Prioritized test cases	gBest	Prioritized test cases	gBest
IP1	1, 6, 5, 20, 16, 12, 0, 19, 10, 2	0.004726	1, 16, 0, 32, 30, 9, 2, 26, 5, 35	0.000465	43, 16, 1, 0, 27, 47, 20, 23, 22, 6	0.000991
IP2	45, 16, 44, 15, 3, 13, 18, 25, 1, 30	0.009869	0, 28, 43, 4, 29, 21, 20, 18, 19, 25	0.009865	48, 28, 27, 43, 3, 35, 33, 14, 1,24	0.009865
IP3	28, 49, 45, 4, 18, 43, 8, 41, 44, 38	0.000003	11, 5, 7, 4, 48, 34, 26, 49, 42, 44	0.000004	6, 38, 37, 39, 47, 7, 15, 11, 36, 0	0.001366
IP4	39, 38, 47, 3, 36, 23, 16, 34, 42, 7	0.000015	1, 5, 35, 8, 4, 27, 31, 42, 32, 3	0.001010	38, 13, 8, 25, 49, 18, 12, 6, 4, 23	0.009917
IP5	26, 49, 48, 9, 5, 47, 2, 25, 32, 23	0.001612	12, 26, 22, 40, 27, 34, 7, 14, 47, 1	0.002611	45, 40, 2, 5, 49, 31, 38, 1, 41, 21	0.001228
IP6	30, 18, 39, 21, 49, 14, 4, 2, 37, 27	0.000010	4, 45, 42, 36, 39, 43, 21, 31, 49, 33	0.000611	42, 43, 9, 46, 25, 37, 41, 35, 0, 5	0.001267
Time (ms)	52		8		6	

6 Results and Interpretation

In order to prove the effectiveness of PSO with the specified BFs we ran PSO on fifteen JAVA programs under same test environment to produce unbiased results. In Table 7 we have listed the results derived during this work in terms of the time taken by the algorithm and the global best value reached for every independent path. For implementation simplicity we have shown the value for global best value up to five decimal places for all the identified paths in the program. The last column of the table i.e., the time taken for prioritization is calculated by addition of time taken for prioritizing each independent path for the program. The least time taken by PSO amongst the three functions is highlighted as bold in the table and where a path doesn't exist the entry is shown as **X**.

The time taken by Griewank is approximately 8.90% of the average time taken by Sphere function and 64.17% of the average time taken by Rastrigin function. The results prove that Griewank works best amongst the three functions with PSO by taking into consideration the execution time and value of gBest achieved in comparison to the optimal value of the benchmark functions as specified in Table 2.

Table 7 Results (Time taken by the algorithm) and the global best value for every independent path

P	BF	Global best (gBest)						Mean time
		Path 1	Path 2	Path 3	Path 4	Path 5	Path 6	
P1	Sphere	0.00231	0.00075	0.00000	0.00254	0.00115	0.00042	59
	Rastrigin	0.00995	0.00986	0.00000	0.00077	0.00161	0.00034	8
	Griewank	0.00988	0.00986	0.00000	0.00992	0.00063	0.00687	**6**
P2	Sphere	0.00236	409.565	9.69788	X	X	X	63
	Rastrigin	0.00996	0.30067	0.00002				4
	Griewank	0.00044	0.42755	0.41175				**3**
P3	Sphere	2.35024	0.07074	0.00686	0.03865	X	X	55
	Rastrigin	0.00314	0.01003	0.00085	0.00336			12
	Griewank	0.05675	0.01127	0.00637	0.01421			**9**
P4	Sphere	0.00001	0.00061	0.00024	X	X	X	49
	Rastrigin	0.00036	0.00060	0.00021				4
	Griewank	0.00002	0.01452	0.00159				**2**
P5	Sphere	0.00001	0.02694	X	X	X	X	47
	Rastrigin	0.00036	0.00222					4
	Griewank	0.00128	0.00055					**2**
P6	Sphere	0.68587	0.00076	X	X	X	X	42
	Rastrigin	0.03897	0.00740					5
	Griewank	0.00987	0.00330					**3**
P7	Sphere	0.02826	0.00869	X	X	X	X	43
	Rastrigin	0.00099	0.00411					**6**
	Griewank	0.08740	0.00019					**4**

(*continued*)

Table 7 (*continued*)

P	BF	Global best (gBest)						Mean time
		Path 1	Path 2	Path 3	Path 4	Path 5	Path 6	
P8	Sphere	0.01021	0.09533	0.00620	0.00073	0.02167	0.00011	50
	Rastrigin	0.02753	0.00848	0.00794	0.01016	0.00005	0.00019	8
	Griewank	0.08981	0.16629	0.03268	0.00710	0.00053	0.00115	**5**
P9	Sphere	0.00018	0.01118	0.02522	X	X	X	38
	Rastrigin	0.00612	0.01024	0.00026				8
	Griewank	0.00990	0.01149	0.01509				**7**
P10	Sphere	0.09492	0.01024	X	X	X	X	43
	Rastrigin	0.02038	0.00714					5
	Griewank	0.00604	0.01190					**2**
P11	Sphere	0.64828	0.02668	0.00068	X	X	X	44
	Rastrigin	0.06147	0.00174	0.00982				**4**
	Griewank	0.01014	0.00000	0.00001				**4**
P12	Sphere	0.00068	5632.89	159.076	X	X	X	49
	Rastrigin	0.04709	7.91535	0.16338				5
	Griewank	0.00077	14.2083	0.78569				**3**
P13	Sphere	0.00015	0.05450	0.00595	X	X	X	50
	Rastrigin	0.00279	0.00298	0.01003				4
	Griewank	0.00001	0.01432	0.01013				**2**
P14	Sphere	0.00204	0.10044	0.00246	0.08367	0.02341		58
	Rastrigin	0.00036	0.01834	0.01393	0.01018	0.00992		7
	Griewank	0.00378	0.00989	0.00042	0.00090	0.01061		**4**
P15	Sphere	0.03100	0.49967	X	X	X	X	40
	Rastrigin	0.00818	0.00747					3
	Griewank	0.01020	0.00146					**2**

The bar graph for the results driven in Table 7 is shown in Fig. 6. In the graph x-axis represents the 15 programs that have been taken in this work and y-axis represent the mean time taken by PSO for prioritizing test cases for the selected programs using three functions: Sphere, Rastrigin and Griewank. The results drawn are not repeatable as time may vary based on system configuration and the programming language on which the experiment is performed.

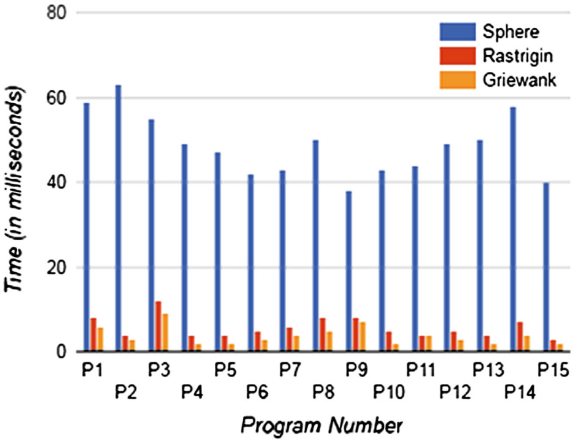

Fig. 6 Time taken by benchmark functions with PSO

7 Threats to Validity

This empirical investigation is carried out on Eclipse that may not be the correct representative for all the software's available in the market but we have tried to the best of our ability to generalize our results for different scenarios. For making our results unbiased and generalized we have followed the basic concepts of object-oriented architecture while implementing fifteen program in JAVA. The results presented in this empirical investigation would not be complete without the discussion of threats of validity: External, Internal and Constraint.

External Validity means the degree of to which the results can be generalized and includes the factor that impact our ability for generalizing the results. The main treat to external validity in this work is that the test programs are small and medium with similar fault patterns and that might not truly representing large scope of programs.

Internal Validity is defined as the degree to measure the consequences on change of independent variables on our dependent variables. In this work, we have minimized this effect by selecting the range of random number between [0–1] for calculation of PSO variables.

Constraint Validity is the degree to which the results are appropriately captured using independent and dependent variables. The way in which random numbers generated for calculating PSO variables may vary depending on the framework used for JAVA programming. In this work, we have tried to minimize this threat using eclipse for JAVA development that is widely used in many organizations for large and complex program.

8 Conclusion and Future Work

In this work, we have used three BFs to calculate fitness of test cases and then sorted test suite based on fitness value. We have verified the effectiveness of Griewank function with PSO in terms of global best value achieved and time required for prioritization of test cases. The results are derived by running fifteen java programs with three BFs and it is proved Griewank works best with PSO in terms of average time required for prioritizing test cases.

In future we plan to run the algorithm for programs with 1000 and more LOC and verify the results under the same test environment. The effectiveness of the approach suggested could we further improve using more BFs or either by combining one or more BFs while calculating fitness of particles in PSO algorithm. The algorithm can also be blended with other meta-heuristic algorithm like GA, MA and ACO. A hybrid PSO can be designed that uses a combination of the three functions for test suite prioritization.

References

1. Joseph, A.K., Radhamani, G.: A hybrid model of particle swarm optimization (PSO) and artificial bee colony (ABC) algorithm for test case optimization. Int. J. Comput. Sci. Eng. (IJCSE) **3**(5) (2011)
2. Suri, B., Singhal, S.: Implementing ant colony optimization for test case selection and prioritization. Int. J. Comput. Sci. Eng. **3**(5), 1924–1932 (2011)
3. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Trans. Software Eng. **27**(10), 929–948 (2001)
4. Mor, M.A.: Evaluate the effectiveness of test suite prioritization techniques using APFD metric. IOSR J. (IOSR Journal of Computer Engineering) **1**(16), 47–51
5. El-Sherbiny, M.M.: Particle swarm inspired optimization algorithm without velocity equation. Egypt. Inf. J. **12**(1), 1–8 (2011)
6. Malhotra, R., Khari, M., Molga. M., Smutnicki, C.: Test suite optimization using mutated artificial bee colony. In: Proceedings of International Conference on Advances in Communication, Network, and Computing, CNC, Elsevier, pp. 45–54 (2014)
7. Hla, K.H.S., Choi, Y., Park, J.S.: Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In: IEEE 8th International Conference on Computer and Information Technology Workshops, 2008. CIT Workshops 2008, pp. 527–532, IEEE, July 2008
8. Oo, N.W.: A comparison study on particle swarm and evolutionary particle swarm optimization using capacitor placement problem. In: Power and Energy Conference, 2008. PEC on 2008. IEEE 2nd International, pp. 1208–1211. IEEE, December 2008
9. Wang, H., Qian, F.: An improved particle swarm optimizer with behavior-distance models and its application in soft-sensor. In: 7th World Congress on Intelligent Control and Automation, 2008. WCICA 2008, pp. 4473–4478. IEEE, June 2008
10. Singla, S., Kumar, D., Rai, H.M., Singla, P.: A hybrid PSO approach to automate test data generation for data flow coverage with dominance concepts. Int. J. Adv. Sci. Technol. **37**, 15–26 (2011)
11. Jamil, M., Yang, X.-S.: A literature survey of benchmark functions for global optimisation problems. Int. J. Math. Modell. Numer. Optimisation **4**(2) (2013)

12. Hassan, R., Cohanim, B., De Weck, O., Venter, G.: A comparison of particle swarm optimization and the genetic algorithm. In: 46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference, p. 1897 (2005)
13. Yang, X.S.: Nature-inspired metaheuristic algorithms. Luniver Press (2010)
14. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Time aware test suite prioritization. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 1–12. ACM, July 2006
15. Sharma, I., Kaur, J., Sahni, M.: A test case prioritization approach in regression testing. Int. J. Comput. Sci. Mob. Comput. **3**, 607–614 (2014)
16. Nayak, N., Mohapatra, D.P.: Automatic test data generation for data flow testing using particle swarm optimization. Contemp. Comput, 1–12 (2010)
17. Chawla, P., Chana, I., Rana, A.: A novel strategy for automatic test data generation using soft computing technique. Frontiers Comput. Sci. **9**(3), 346–363 (2015)
18. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verification Reliab. **22**(2), 67–120 (2012)
19. Kaur, A., Bhatt, D.: Particle swarm optimization with cross-over operator for prioritization in regression testing. Int. J. Comput. Appl. **27**(10) (2011)
20. Kaur, A., Bhatt, D.: Hybrid particle swarm optimization for regression testing. Int. J. Comput. Sci. Eng. **3**(5), 1815–1824 (2011)
21. Kong, X., Sun, J., Xu, W.: Particle swarm algorithm for tasks scheduling in distributed heterogeneous system. In: ISDA'06. Sixth International Conference on Intelligent Systems Design and Applications, 2006, Vol. 2, pp. 690–695. IEEE October 2006
22. Zhi, X.H., Xing, X.L., Wang, Q.X., Zhang, L.H., Yang, X.W., Zhou, C.G., Liang, Y.C.: A discrete PSO method for generalized TSP problem. In: Proceedings of 2004 International Conference on Machine Learning and Cybernetics, 2004, Vol. 4, pp. 2378–2383. IEEE (July–August) 2014
23. Kumar, S., Ranjan, P.: A comprehensive analysis for software fault detection and prediction using computational intelligence techniques. Int. J. Comput. Intell. Res. **13**(1), 65–78 (2017)
24. Hendtlass, T.: Fitness estimation and the particle swarm optimisation algorithm. In: IEEE Congress on Evolutionary Computation, 2007. CEC 2007, pp. 4266–4272. IEEE, September 2007
25. Chen, M., Wang, T., Feng, J., Tang, Y.Y., Zhao, L.X.: A hybrid particle swarm optimization improved by mutative scale chaos algorithm. In: 2012 Fourth International Conference on Computational and Information Sciences (ICCIS), pp. 321–324. IEEE, August 2012
26. Molga, M., Smutnicki, C.: Test functions for optimization needs. http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf