COMPARATIVE EVALUATION OF A MAXIMIZATION AND MINIMIZATION APPROACH FOR TEST DATA GENERATION WITH GENETIC ALGORITHM AND BINARY PARTICLE SWARM OPTIMIZATION

Ankur Pachauri and Gursaran

Department of Mathematics, Dayalbagh Educational Institute, Agra 282110
ankurpachauri@gmail.com, gursaran.db@gmail.com

ABSTRACT
In search based test data generation, the problem of test data generation is reduced to that of function minimization or maximization. Traditionally, for branch testing, the problem of test data generation has been formulated as a minimization problem. In this paper we define an alternate maximization formulation and experimentally compare it with the minimization formulation. We use genetic algorithm and binary particle swarm optimization as the search technique and in addition to the usual operators we also employ a branch ordering strategy, memory and elitism. Results indicate that there is no significant difference in the performance or the coverage obtained through the two approaches and either could be used in test data generation when coupled with the branch ordering strategy, memory and elitism.

KEYWORDS
Search based test data generation, program test data generation, genetic algorithm, software testing

1. INTRODUCTION
Search-based software test data generation has emerged [1, 2, 3, 4, 5, 6] as a significant area of research in software engineering. In search based test data generation, the problem of test data generation is reduced to that of function minimization or maximization. The source code is instrumented to collect information about the program as it executes. Collected information is used to heuristically measure how close the test data is to satisfying the test requirements. The measure is then used to modify the input parameters to progressively move towards satisfying the test requirement. It is here that the application of metaheuristic search techniques has been explored. Traditionally, for branch testing, the problem of test data generation has been formulated as a minimization problem. In this paper we define an alternate maximization formulation and experimentally compare with the traditional minimization formulation.

During testing, program P under test is executed on a test set of test data - a specific point in the input domain - and the results are evaluated. The test set is constructed to satisfy a test adequacy criterion that specifies test requirements [7, 8]. The branch coverage criterion is a test adequacy criterion that is based on the program flow graph. More formally, a test set T is said to satisfy the branch coverage criterion if on executing P on T, every branch in P’s flow graph is traversed at least once.

Metaheuristic techniques such as genetic algorithms [9], quantum particle swarm optimization [10], scatter search [11] and others have been applied to the problem of automated test data generation and provide evidence of their successful application. Amongst these several have addressed the issue of test data generation with program-based criteria [10] and in particular the branch coverage criterion [10, 11, 12, 13, 14, 15, 16, 17, 18]. Further, [12, 13, 14, 19, 20] have formulated the problem as a minimization problem. In this paper we consider an alternate
maximization formulation and compare it with the minimization strategy. We use both Genetic Algorithm and the Binary Particle Swarm Optimization as the search strategy.

In this paper, in Section 2 we describe the Genetic Algorithm (GA) and in Section 3 we describe the Particle Swarm Optimization (PSO) and Binary Particle Swarm Optimization (BPSO). We outline the basic maximization and minimization strategies for test data generation in Section 4 and also describe the other strategies employed by us. In Section 5 we present the experimental setup and in Section 6 we discuss the results of the experiments. Section 7 concludes the paper.

2. GENETIC ALGORITHM

Genetic Algorithm (GA) is a search algorithm that is based on the idea of genetics and evolution in which new and fitter set of string individuals are created by combining portions of fittest string individuals of the parent population [21]. A genetic algorithm execution begins with a random initial population of candidate solutions \{s_i\} to an objective function f(s). Each candidate s_i is generally a vector of parameters to the function f(s) and usually appears as an encoded binary string (or bit string) called a chromosome or a binary individual. An encoded parameter is referred to as a gene, where the parameter’s values are the gene’s alleles. If there are m inputs parameters with the ith parameter expressed in n_i bits, then the length of the chromosome is simply \(\sum_i n_i\). In this paper each binary individual, or chromosome, represents an encoding of test data.

After creating the initial population, each chromosome is evaluated and assigned a fitness value. Evaluation is based on a fitness function that is problem dependent. From this initial selection, the population of chromosomes iteratively evolves to one in which candidates satisfy some termination criteria or, as in our case, fail to make any forward progress. Each iteration step is also called a generation.

Each generation may be viewed as a two stage process [21]. Beginning with the current population, selection is applied to create an intermediate population and then recombination and mutation are applied to create the next population. The most common selection scheme is the roulette-wheel selection in which each chromosome is allocated a wheel slot of size in proportion to its fitness. By repeatedly spinning the wheel, individual chromosomes are chosen using “stochastic sampling with replacement” to construct the intermediate population. Additionally with elitism the fittest chromosomes survive from one population to the other.

After selection, crossover, i.e., recombination, is applied to randomly paired strings with a probability. Amongst the various crossover schemes are the one point, two point and the uniform crossover schemes [21]. In the one point case a crossover point is identified in the chromosome bit string at random and the portions of chromosomes following the crossover point, in the paired chromosomes, are interchanged. In addition to crossover, mutation is used to prevent permanent loss of any particular bit or allele. Mutation application also introduces genetic diversity. Mutation results in the flipping of bits in a chromosome according to a mutation probability which is generally kept very low.

The chromosome length, population size, and the various probability values in a GA application are referred to as the GA parameters in this paper. Selection, crossover, mutation are also referred to as the GA operators.
3. PARTICLE SWARM OPTIMIZATION AND BINARY PARTICLE SWARM OPTIMIZATION

Particle Swarm Optimization (PSO) was initially proposed to find optimal solutions for continuous space problems by Kennedy and Eberhart [22, 23] in 1995. In PSO the search starts with a randomly generated population of solutions called the swarm of particles in \( d \)-dimensional solution space. Particle \( i \) is represented as \( X_i = (x_{i1}, x_{i2}, \ldots, x_{id}) \) which is called the position of the particle \( i \) in \( d \)-dimensional space. With every particle \( i \) a velocity vector \( V_i = (v_{i1}, v_{i2}, \ldots, v_{id}) \) is associated that plays an important role in deciding next position of particle and is updated in each iteration. For updating the velocity of each particle, the particle’s best \( P_{ibest} = (p_{i1}, p_{i2}, \ldots, p_{id}) \) which is the best position of particle \( i \) achieved so far and global best \( P_{gbest} = (p_{g1}, p_{g2}, \ldots, p_{gd}) \) which is the best position of the swarm achieved so far by any particle of the swarm, are used.

Following equations (1) and (2) are used to find new velocity and position of particle \( i \) in iteration \( t+1 \).

\[
V_i(t+1)=w \cdot V_i(t)+c_1 \cdot \phi_1 \cdot (p_{ibest}-X_i(t))+c_2 \cdot \phi_2 \cdot (p_{gbest}-X_i(t)) \tag{1}
\]

\[
X_i(t+1)=X_i(t)+V_i(t+1) \tag{2}
\]

In equation (1), \( w \) is the inertia weight which controls the impact of previous history of velocity on global and local search abilities of particles [23], \( c_1 \) and \( c_2 \) are positive learning constants which determine the rate by which the particle moves towards individual’s best position and the global best position. Usually, \( c_1 \) and \( c_2 \) are chosen in a way so that their sum doesn’t exceed 4. If it exceeds 4 at any time then both the velocities and positions will explode toward infinity. \( \phi_1 \) and \( \phi_2 \) are random numbers drawn from uniform probability distribution of \((0, 1)\). In this way positions and velocities of the particles are evolved in each iteration until the optimal solution is not obtained.

In 1997 Kennedy and Eberhart [24] introduced the binary particle swarm optimization (BPSO) algorithm. In the binary version every particle is represented by a bit string and each bit is associated with a velocity, which is the probability of changing the bit to 1. Particles are updated bit by bit and velocity must be restricted within the range \([0, 1]\). Let \( P \) be the probability of changing a bit from 0 to 1, then \( 1-P \) will be the probability of not changing the bit to 1. This probability can be represented as the following function:

\[
P(x_{id}(t) = 1) = f(x_{id}(t), v_{id}(t-1), p_{ad}, p_{gd}) \tag{3}
\]

where \( P(x_{id} = 1) \) is the probability that an individual particle \( i \) will choose 1 for the bit at the \( d \)-site in the bit string, \( x_{id}(t) \) is the current state of particle \( i \) at bit \( d \), \( v_{id}(t-1) \) is a measure of the string’s current probability to choose a 1, \( p_{ad} \) is the best state found so far for bit \( d \) of individual \( i \), i.e., a 1 or a 0, \( p_{gd} \) is 1 or 0 depending on what the value of bit \( d \) in the global best particle.

The most commonly used measure for \( f \) is the sigmoid function which is defined as follows:

\[
f(v_{id}(t)) = \frac{1}{1 + e^{-v_{id}(t)}} \tag{4}
\]

where,

\[
v_{id}(t)=w_{id}v_{id}(t-1)+(\varphi_1)(p_{ad}-x_{id}(t-1))+(\varphi_2)(p_{gd}-x_{id}(t-1)) \tag{5}
\]

Equation (5) gives the update rule for the velocity of each bit, where \( \varphi_1 \) and \( \varphi_2 \) are random numbers drawn from the uniform distributions. Sometimes these parameters are chosen from the uniform
distribution, such that their sum is 4. The value is sometimes limited so that does not approach 0.0 or 1.0 too closely. In this case, constant parameters \([V_{\text{min}}, V_{\text{max}}]\) are used. When \(V_{\text{id}}\) is greater than \(V_{\text{max}}\), it is set to \(V_{\text{max}}\) and if \(V_{\text{id}}\) is smaller than \(V_{\text{min}}\), then \(V_{\text{id}}\) is set to \(V_{\text{min}}\). This simply limits the ultimate probability that bit \(x_{\text{id}}\) will take on a zero or one value. A higher value of \(V_{\text{max}}\) makes new vectors less likely. Thus \(V_{\text{max}}\) in the discrete particle swarm plays the role of limiting exploration after the population has converged [24], i.e., it can be said that \(V_{\text{max}}\) controls the ultimate mutation rate or temperature of the bit vector. Smaller \(V_{\text{max}}\) leads to a higher mutation rate [24]. This is explored in the experiment described in this paper.

4. TEST DATA GENERATION FOR BRANCH COVERAGE

In search based test data generation, test data is generated to meet the requirements of a particular test adequacy criterion. The criterion in our case is the branch coverage criterion. The setup phase begins with the choice of a suitable representation for test data and the identification of a suitable fitness function.

The inputs for one execution of the program under test \(P\), i.e., a single test data, are represented in a binary string also called a \textit{binary individual}. For instance, if the input to \(P\) is a pair of integers 

ist represented in a bit sequence of length \(\text{rep}(I_1)+\text{rep}(I_2)\) where \(\text{rep}(x)\) is the number of bits taken to represent \(x\). The length of the bit strings representing \(I_1\) and \(I_2\) are chosen to represent the largest legal value that can be input to \(P\). In the representation, the bit sequence representing \(I_1\) is followed by the bit sequence representing \(I_2\).

The fitness of a binary individual is computed as

\[
\text{Fitness}(x) = \text{Approximation Level} + \text{Normalized Branch Distance}
\]

Traditionally, test data generation problem is formulated as a minimization problem as in [12, 13, 14, 19, 20] in which the approach level numbering starts from the target branch and the normalized branch distance is computed as,

\[
\text{Normalized Branch Distance} = 1 - 1.001^{\text{distance}}
\]

As opposed to this, the test data generation problem can also be formulated as a maximization problem. The definition of approximation level and normalized branch distance is also different from [2] although the basic idea is similar.

The \textit{approximation Level} is a count of the number of predicate nodes in the shortest path from the first predicate node in the flow graph to the predicate node with the \textit{critical branch} - a branch that leads the target to be missed in a path through the program- as shown in Figure 1.

The \textit{Normalized Branch Distance} is computed according to the formula

\[
\text{Normalized Branch Distance} = (1/(1.001^{\text{distance}}))
\]

where, \textit{distance}, or \textit{branch distance}, as defined in [20, 25], is computed at the node with the critical branch using the values of the variables or constants involved in the predicates used in the conditions of the branching statement. Table 1 summarizes the computation of distance.

<table>
<thead>
<tr>
<th>Decision Type</th>
<th>Branch Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a &lt; b)</td>
<td>(a - b)</td>
</tr>
<tr>
<td>(a \leq b)</td>
<td>(a - b)</td>
</tr>
<tr>
<td>(a &gt; b)</td>
<td>(b - a)</td>
</tr>
<tr>
<td>(a \geq b)</td>
<td>(b - a)</td>
</tr>
<tr>
<td>(a == b)</td>
<td>(\text{Abs}(a - b))</td>
</tr>
<tr>
<td>(a != b)</td>
<td>(\text{Abs}(a - b))</td>
</tr>
<tr>
<td>(a &amp;&amp; b)</td>
<td>(a + b)</td>
</tr>
<tr>
<td>(a</td>
<td></td>
</tr>
</tbody>
</table>

Entries one through five are the same as in [19]. Table 1 also describes the computation of distance in the presence of logical operators AND (&&) and OR (||). In both these cases, the definition takes into account that branch distance is to be minimized whereas the fitness is to be maximized.

Figure 1. Approximation Level and Branch Distance Computation

In general, in order to generate test data to satisfy the branch coverage criterion using GA and BPSO, the sequence in which the branches will be selected for coverage must be defined. A chosen branch may become difficult to cover if the corresponding branch predicate is not reached by any of the test data or individuals in the current population. One of the proposals made by Pachauri and Gursaran [26] for sequencing is the path prefix strategy. We adopt this strategy for the experiments described in this paper. Further, each time a branch is traversed for the first time, it may be necessary to store the test data that traverse the branch and inject these into the population when the sibling branch is selected for traversal. This is referred to as memory and is used in this paper. In order to ensure that individuals reaching the sibling branch of the target are not destroyed by the genetic algorithm operators, elitism is adopted. Up to 10% of fit individuals, with a minimum of one individual, are carried forward to the next generation. Furthermore, it is also possible to initialize the population each time a new branch is selected for coverage or leave it uninitialized. In the experiments described in this paper, the population is not initialized.

Infeasibility may prevent test data from being generated to satisfy a coverage criterion. It may be dealt with as follows. If the search is attempting to traverse a particular branch, but is unable to do so over a sufficiently large, predetermined, number of iterations, then the search run is aborted and the branch is manually examined for infeasibility. If the branch is found to be infeasible then it is marked as traversed and the search is rerun.
5. **EXPERIMENTAL SETUP**

In this section we describe the various experiments carried out to test the performance of test data generation with genetic algorithm.

5.1 Benchmark Programs

Benchmark programs chosen for the experiments have been taken from [11, 27]. These programs have a number of features such as real inputs, equality conditions with the AND operator and deeply nested predicates that make them suitable for testing different approaches for test data generation.

- **Line in a Rectangle Problem**: This program takes eight real inputs, four of which represent the coordinates of rectangle and other four represents the coordinates of the line. The program determines the position of the line with respect to the position of rectangle and generates one out of four possible outputs:
  
  A. The line is completely inside the rectangle;
  B. The line is completely outside the rectangle;
  C. The line is partially covered by the rectangle; and
  D. Error: The input values do not define a line and/or a rectangle.

  The maximum nesting level is 12. In total this program’s CFG has 54 nodes and 18 predicate nodes.

- **Number of Days between Two Dates Problem**: This program calculates the days between two given dates of the current century. It takes six integer inputs- three of which represent the first date (day, month, and year) and other three represents the second date (day, month, and year). The CFG has 43 predicate nodes and 127 nodes

- **Calday**: This routine returns the Julian day number. There are three integer input to the program. First input represent month, second represent day and the third represent the year. It's CFG has 27 Nodes with 11 predicate nodes. It has equality conditions, remainder operator. The maximum nesting level is 8.

- **Complex Branch**: It accepts 6 short integer inputs. In this routine there are some complex predicate conditions with relational operators combined with complex AND and OR conditions, it also contains while loops and SWITCH-CASE statement. Its CFG contains 30 nodes.

- **Meyer’s Triangle Classifier Problem**: This program classifies a triangle on the basis of its input sides as non triangle or a triangle, i.e., isosceles, equilateral or scalene. It takes three real inputs all of which represent the sides of the triangle. It's CFG has 14 Nodes with 6 predicate nodes. The maximum nesting level is 5. It has equality conditions with AND operator, which make the branches difficult to cover.

- **Sthamer’s Triangle Classifier Problem**: This program also classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral, right angle triangle or scalene. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It's CFG has 29 Nodes with 13 predicate nodes. The maximum nesting level is 12. It has equality conditions with AND operator and complex relational operators.

- **Wegener’s Triangle Classifier Problem**: This program also classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral, orthogonal or obtuse angle. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It’s CFG has 32 Nodes with 13 predicate nodes.

- **Michael’s Triangle Classifier Problem**: This program also classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral or scalene. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It's CFG has 26 Nodes with 11 predicate nodes. The maximum nesting level is 6.

### 5.2 GA Operator and Parameter Settings

Table 2 lists the various operator and parameter settings for the genetic algorithm used in this study.

<table>
<thead>
<tr>
<th>Parameter/ Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Population Size</td>
<td>6, 10, 16, 20, 26, …, 110.</td>
</tr>
<tr>
<td>2 Crossover type</td>
<td>Two point crossover</td>
</tr>
<tr>
<td>3 Crossover Probability</td>
<td>1.0</td>
</tr>
<tr>
<td>4 Mutation Probability</td>
<td>0.01</td>
</tr>
<tr>
<td>5 Selection Method</td>
<td>Binary tournament</td>
</tr>
<tr>
<td>6 Branch Ordering Scheme</td>
<td>Path Prefix Strategy</td>
</tr>
<tr>
<td>7 Fitness Function</td>
<td>As described in Section 3.</td>
</tr>
<tr>
<td>8 Population Initialization</td>
<td>Initialize once at the beginning of the GA run</td>
</tr>
<tr>
<td>9 Population Replacement Strategy</td>
<td>Elitism with upto 10% carry forward</td>
</tr>
<tr>
<td>10 Maximum Number of Generations</td>
<td>10⁷</td>
</tr>
<tr>
<td>11 Memory</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3 lists the various operator and parameter settings for the Binary Particle Swarm Optimization (BPSO) used in this study.

<table>
<thead>
<tr>
<th>Parameter/ Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ϕ₁, ϕ₂</td>
<td>Random numbers from the uniform distribution (0,4), such that ϕ₁ + ϕ₂ ≤ 4</td>
</tr>
<tr>
<td>w</td>
<td>[0.5+ (rnd/2.0)], where rnd is random number drawn from uniform distribution (0,1).</td>
</tr>
</tbody>
</table>

### 6. RESULTS

Experiments with the two approaches were carried out and compared independently for Genetic Algorithm and for Binary Particle Swarm Optimization. For each population size, hundred experiments were carried out and the following statistics were collected:

- Mean number of generations. It may be noted that the termination criterion for each experiment is either full branch coverage or 10⁷ generations whichever occurs earlier. The number of generations to termination over hundred experiments is used to compute the mean. The mean does not tell us if all the branches were covered.

- Mean percentage coverage achieved.
Additionally ANOVA was carried out using SYSTAT 9.0 to determine significant difference in means for experiments with Genetic Algorithm only.

In all the experiments with GA and BPSO, full (100%) coverage was achieved for all population sizes, for all benchmark programs and for both maximization and minimization approaches. This implies that the differentiating factor would have to be the difference in the mean number of generations.

Figure 2 and Figure 3 plot the mean number of generations for both the maximization and minimization approach with Genetic Algorithm. Table 4 summarizes the results of ANOVA with F and p values for the GA based results. Considering a significance level of 0.05, it can be seen that the difference for all the benchmark programs is not significant except for some isolated cases which are not generalizable.

Similar results are also obtained for BPSO, which can also be seen in Figure 4 through Figure 5. Results with both the approaches are comparable.

Further analysis in our case shows that with the path prefix strategy and memory, individuals are present in each generation that cause a traversal of the sibling branch of the target. This coupled with elitism may actually speed up the test data discovery process.
Figure 3  Plots of Mean Number of Generations for Benchmark Programs using Genetic Algorithm

Table 4. Results of test of ANOVA using Genetic Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Calday Program</th>
<th>Complex-Branch Program</th>
<th>Sthamer-Triangle Program</th>
<th>Wegener-Triangle Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>F value</td>
<td>P value</td>
<td>F value</td>
<td>P value</td>
<td>F value</td>
</tr>
<tr>
<td>10</td>
<td>0.095</td>
<td>0.762</td>
<td>0.003</td>
<td>0.003</td>
</tr>
<tr>
<td>20</td>
<td>0.028</td>
<td>0.866</td>
<td>0.869</td>
<td>0.003</td>
</tr>
<tr>
<td>30</td>
<td>0.264</td>
<td>0.608</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>40</td>
<td>0.679</td>
<td>0.411</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>50</td>
<td>1.311</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>60</td>
<td>1.647</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>70</td>
<td>2.127</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>80</td>
<td>3.335</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>90</td>
<td>4.004</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>100</td>
<td>5.308</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>110</td>
<td>6.786</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>120</td>
<td>8.711</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>130</td>
<td>11.309</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>140</td>
<td>13.716</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>150</td>
<td>16.907</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
</tbody>
</table>
Figure 4  Plots of Mean Number of Generations for Benchmark Programs using Binary Particle Swarm Optimization (BPSO)

Figure 5  Plots of Mean Number of Generations for Benchmark Programs using Binary Particle Swarm Optimization (BPSO)
7. CONCLUSION

In search based test data generation, the problem of test data generation is reduced to that of function minimization or maximization. Traditionally, for branch testing, the problem of test data generation has been formulated as a minimization problem. In this paper we have defined an alternate maximization formulation and experimentally compared it with the minimization formulation. We have used a genetic algorithm and binary particle swarm optimization as the search technique and in addition to the usual operators we have also employed the path prefix strategy as a branch ordering strategy and memory and elitism. Results indicate that there is no significant difference in the performance or the coverage obtained through the two approaches and either could be used in test data generation if coupled with the path prefix strategy, memory and elitism.

ACKNOWLEDGEMENTS

This work was supported by the UGC Major Project Grant F.No.36-70/2008 (SR) for which the authors are thankful.

REFERENCES


