Software Test Data Generation using Path Prefix Strategy and Genetic Algorithm

Ankur Pachauri
Department of Mathematics
Dayalbagh Educational Institute
Agra, India
+915622801545
ankurpachauri@gmail.com

Gursaran
Department of Mathematics
Dayalbagh Educational Institute
Agra, India
+915622801545
gursaran.db@gmail.com

ABSTRACT
The Path Prefix Strategy for test data generation suggested by Prather and Myers [13] can be used to order branches for selection for coverage in a test data generation system using the Genetic Algorithm (GA). This is described in this paper. Further, in order to get over the problems associated with the use of the simple genetic algorithm (SGA) for test data generation with the proposed approach for branch ordering, elitism, elitism with fitness scaling and elitism with fitness scaling and memory were considered. These have been evaluated with extensive experiments on benchmark programs and the results of these experiments are presented and discussed. An experimental comparison with a GA implementation that did not incorporate the proposed branch ordering approach, elitism and memory, seems to indicate that a combination of the three may result in much better performance.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging – testing strategies, test design and test coverage of code.

General Terms
Verification

Keywords
Software test data generation, genetic algorithm.

1. INTRODUCTION
A conventional, structural test data generation methodology can be divided into three phases [4]:
1. Program Analysis: Different representations such as control flow graph, data flow graph and other objects of interest are extracted from program code.
2. Path Selection: A finite set of program paths are chosen to satisfy a chosen test data adequacy criterion.
3. Test Data Generation: Test data is generated to cover the selected paths.

In order to generate test data, an inverse problem [13] – given a path find a program input that traverses the path - is solved. This problem may become difficult in the presence of infeasible paths and may also necessitate an iteration between the path selection and test data generation phases above. In an attempt to circumvent this, Prather and Myers [13] suggested the use of an adaptive strategy in which one new test path, or sub-path, is added at a time and previous paths serve as a guide for selection of subsequent paths using an inductive strategy. The path prefix strategy suggested by them is one such strategy for test data generation.

The path prefix strategy [13] defines a procedure to order branches, through the identification of path prefixes, for selection for coverage. For a path \( p \) that is traversed in an execution, a reversible prefix \( q \) is defined as the minimal initial portion of path \( p \) to a decision node \( d \), whose branches are not fully covered, and the branch \( b \) that is covered by \( p \). This is illustrated in Figure 1. At any stage if \( p_1, ..., p_k \) are the traversed paths, the idea is to find an input \( x_k \) to cause the reversal of shortest reversible prefix \( q \), with reversal \( q' \), amongst all the \( p_i \)’s.

Genetic algorithms can take advantage of the path prefix strategy by making available a pool of reversible path prefixes, since a number of paths are traversed in parallel.

Further, if we select a reversible prefix \( q \) out of the pool such that the largest number of chromosomes in the current population encode test cases that traverse \( q \), then we can exploit the properties of evolution in the genetic algorithm to evolve a test case (chromosome) that covers the untraversed branch \( b' \) in its reversal \( q' \). It may also be noted that we may require just the branch \( b' \) to be covered and not the subpath \( q' \).

In general, metaheuristic techniques have now been widely used in software test data generation. Techniques such as simulated
With probability $7$, repeat {  // generate a new population

An application of metaheuristic techniques for test data generation is usually based on the idea of dynamic test data generation methodology proposed by Korel [5, 7]. During dynamic test generation, the source code of a program is instrumented to collect information about the program as it executes. The resulting information, collected during each test execution of the program, is used to heuristically determine how close the test case is to satisfying the test requirement as specified by the selected test criterion. This allows the test generator to modify the program's inputs gradually, moving them ever closer to values that satisfy the requirement. In other words, the problem of generating test data reduces to the problem of function minimization (or maximization) [1].

In this paper we have attempted to use the path prefix strategy for generating test cases for a given program with the Genetic Algorithm (GA). The strategy is used for generating a branch ordering for coverage. However, the poor performance of a simple GA (SGA) [6] in pilot experiments led us to experiment with the following schemes also:

(a) SGA with elitism (EGA).
(b) SGA with fitness scaling (EGAFS).
(c) EGA with fitness scaling and memory (EGAFSM).

These we describe in the subsequent sections along with the experimental results.

In Section 2 we outline the application of a Genetic Algorithm for test data generation and describe the schemes mentioned in the previous paragraph. In Section 3 we describe the experiment and the benchmark programs. In Section 4 we present the results. In order to study the effect of mutation probability, in section 4, we also present an experiment with different mutation probabilities for some chosen GA parameters. In Section 5 we try evaluate the improvement in performance with EGAFSM over a GA implementation that does not use the proposed branch ordering, elitism and memory. Section 6 concludes the paper.

In dynamic 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. A GA setup begins with the choice of a chromosome representation and the identification of a suitable fitness function. A chromosome in our case represents a test case, i.e., the inputs to the program. With branch coverage, raw fitness is evaluated based on the predicates in branching conditions. Assuming that fitness is being maximized, for each branch, chromosome fitness is evaluated as: $\text{Fitness} = 1/ |h - g + \delta|$, where $\delta$ is a small quantity (approximately 0.0001) to prevent numeric overflow, $h$ and $g$ represent the desired and the actual branch predicate values respectively. Here, if the decision node is not reached, the fitness is assumed to be very low. Program $P$ is suitably instrumented to give $P_t$ so as to mark the coverage of branches as they are traversed in an execution of $P_t$ and to return a computed fitness value for the test case.

In step 3a, we implement the simple genetic algorithm (SGA) as described in Figure 2.

1. Choose an appropriate test adequacy criterion. This in our case is the branch coverage criterion.
2. Setup the GA.
   a. Select a chromosome representation for test case to be input to program $P$.
   b. Select a fitness function.
   c. Instrument the program $P$ to create program $P_t$. The instrumented program $P_t$ is used directly for test data generation.
   d. Select suitable GA parameters.
3. Generate test data.
   a. Run the GA for test data generation using $P_t$ for fitness computation. The Genetic Algorithm is applied at this step.
   b. Identify and eliminate infeasibility.
   c. Regenerate test data if necessary.

In this paper we have attempted to use the path prefix strategy for branch ordering for test data generation with metaheuristic techniques was not seen.

2. GENETIC ALGORITHM AND TEST DATA GENERATION

Let $P$ be the program under test, then the basic sequence of steps in an application of the genetic algorithm for test data generation are as follows:

1. Start with a randomly generated population of $n$ chromosomes (i.e., test cases for $P$)
2. While (termination criterion is not met) {
   3. Execute $P$, on each chromosome (test case) $x$ in the current population;
   4. If (previously selected target branch $b'$ is traversed or if no target has been selected)
      5. Use the path prefix based strategy to identify a new target branch $b''$;
   6. Calculate the fitness $F(x)$ of each chromosome $x$ in the population with respect to target $b''$;
   7. Repeat { if new population
      8. Select a pair of parent chromosomes from the current population;
      9. With probability $P_c$ (the crossover probability), crossover the parents to form two offspring (or children);
      10. Mutate the two offspring with probability $P_m$ (the mutation probability), and place the resulting chromosomes in the new population;
   11. } until (n offspring have been created);
   12. Replace the current population with the new population.
   13. }

Figure 2. The Simple Genetic Algorithm Implementation
The path prefix strategy is used for identifying the target branch for which the test case is to be generated. Here, a potential target branch $b'$ is first selected based on the program paths traversed by the current population. This means that the corresponding decision node $d$ is reached by some test case in the population. However, if no reversible prefix can be selected, a reversible prefix is chosen based on the paths traversed in the previous generations.

In the path prefix strategy, one tries to find an input to the program which will cause the reversal of the shortest reversible prefix. However, in our case, a reversible prefix is finally selected using an additional criterion: the number of chromosomes (test cases) that traverse the reversible prefix. The rule for selecting a reversible prefix and hence a reversal is thus: select the shortest possible reversible prefix with the largest number of test cases traversing it in the current population. This is done to ensure that a sufficient number of test cases exist that reach the decision node where the reversal is sought. Otherwise the fitness of most of the chromosomes with respect to the reversal would be at the lowest value.

In a pilot experiment that was performed with SGA using the path prefix strategy, however, the following problems were observed:

1. At times, the number of test cases reaching the decision node in the selected reversible prefix may be very small and as a result of the crossover and mutation operations, chromosomes (test cases) may be generated such that no test case reaches the decision node, i.e., the fitness becomes zero for all chromosomes.

2. Few superfit, but not optimal, chromosomes tend to dominate and the whole population converges to some local optima.

3. It is possible that a decision node that was traversed earlier is no longer traversed in the current generation and if that node defines a reversal that must be selected, then the fitness of all the chromosomes becomes zero with respect to that reversal.

In order to circumvent these problems, we decided to use elitism to deal with problem (1) above, elitism and fitness scaling to deal with problem (2) and elitism, fitness scaling and memory to deal with problem (3). These are explained in the subsequent paragraphs.

In elitism [11], a percentage of the fittest individuals, i.e., those that reach the decision node $d$ are copied into the next generation. In our case, 10% or one (in case of small population sizes), whichever is greater, is the number of chromosomes copied into the next generation.

Fitness scaling is generally adopted to avoid premature convergence in early stages of a GA run and to amplify the differences in fitness in the later stages of a GA run [15]. In this work, we have chosen sigma truncation scaling [11]. This scaling scheme was chosen to circumvent the problem of negative scaled fitness during later stages of a run. In this scheme, population variance is used to preprocess raw fitness values prior to scaling.

In the selection of reversible prefixes, it is possible that a reversible prefix $q$ is selected with no test case in the population that traverses it. In this situation, we backtrack to the closest decision node which is reached by the current population and from where the flow can be diverted to the decision node $d$ with target branch $b'$. As mentioned earlier this situation would occur if we choose a reversible prefix using a path that was traversed in some previous generation and in which the decision node $d$ with the target branch $b'$ is not reached in the current generation.

In order to circumvent problem (3) above, we make use of memory along with elitism and fitness scaling. Up to a maximum of five test cases are saved for each branch that is traversed in the course of the GA runs. So now when a target branch $b'$ is selected, test cases (chromosomes) that traverse the sibling branch $b$ are introduced into the population.

3. EXPERIMENTS

Extensive experiments were carried out on benchmark programs to evaluate and compare the performance of the following:

(a) Simple Genetic Algorithm (SGA) as described in Figure 2, i.e., with the branch selection strategy described in Section 2.

(b) SGA with elitism (EGA).

(c) EGA with fitness scaling (EGAFS).

(d) EGA with fitness scaling and memory (EGAFSM).

Data was collected for the following parameters settings:

(a) Population size: 6, 10, 16, 20, 26, ….110.

(b) Crossover probability – 1.

(c) Crossover type – Two point crossover.

(d) Mutation probability ($P_m$) was varied from 0.001 through 0.95.

(e) Maximum number of generations – 50,000.

In order to collect data for an adequate statistical comparison, data was collected over 100 runs for each combination of parameter settings.

The experiments were carried out on three benchmark programs, taken from [3], with integer inputs. These are described briefly below:

**Triangle Classifier Problem**

This program classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral or scalene. This program has 12 branches. The maximum nesting level is 5.

**Line in a Rectangle Problem**

This program takes eight integer inputs, four of which represent twenty 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:

1. The line is completely inside the rectangle;
2. The line is completely outside the rectangle;
3. The line is partially covered by the rectangle; and
4. Error: The input values do not define a line and/or a rectangle.

The maximum nesting level is 12. In total this program has 36 branches and 18 statement nodes.
Number of Days between Two Dates Problem

This program calculates the days between two given dates of the current century. It takes six inputs - three of which represent the first date (day, month, and year) and other three represents the second date (day, month, and year). It has 86 branches. As [3] note, this program includes a number of branches with equality conditions. Some of them use the remainder operator (%), which adds discontinuity to the decisions domains and therefore a greater difficulty in finding the tests that cover those branches. The nesting level is very high for most of the branches and, in combination with the AND decisions, the equality conditions and the use of the remainder operator, make this program very appropriate, because of its difficulty, to evaluate the effectiveness and efficiency of an automatic test generator for the branch coverage criterion.

4. RESULTS

4.1 Triangle program
Figure 3 shows graphs for average number of generations vs. population size (see Figures 3a, 3c, 3e, 3g) and average percentage coverage vs. population (see Figures 3b, 3d, 3f, 3h) for SGA, EGA, EGAFS and EGAFSM for the triangle classification problem for four different mutation probabilities of 0.005, 0.01, 0.05 and 0.1. In general, it can be seen that as the mutation probability and population size increases, the average number of generations reduce and the average coverage increases. Both, decrease in the average number of generations and increase in the average coverage indicate that in increasing number of runs, complete coverage could be achieved. However, for mutation probability 0.005 (see Figure 3b), complete coverage could not be achieved in most of the runs for any of the population sizes for which the experiment was carried out. It has been our observation [1], [12], that mutation may play a significant role in helping achieve complete coverage. As the mutation probability is increased, the coverage also improves (see Figures 3b, 3d, 3f and 3h). This seems to suggest that as the exploratory capabilities of the system improve, rather than the exploitative capabilities, the coverage also improves. However, the average number of generations may not continually reduce with increase in mutation probabilities. This is discussed with an experiment in Section 4.4. Here in Figure 3f, we observe that as the mutation probability is set to 0.05, average branch coverage becomes 100% for population size 50 and above with all strategies. In fact, 100% coverage was achieved in all experiments with all population sizes for each of SGA, EGA, EGAFS and EGAFSM with mutation probability equal to 0.1. Figure 3h, shows the average coverage graph for mutation probability 0.1.

In general, as we will also see with other benchmark programs, with low mutation probability, e.g., 0.005, and small population, the performance of EGAFSM, both in terms of average number of generations and average coverage is much superior to SGA and EGA. Surprisingly, one hundred percent average coverage could also be achieved with population sizes 10 and 16 in the triangle case (see Figure 3b). Memory, it seems, may help in improving performance. This is also observed with higher mutation probabilities. However, with increasing mutation probability, the performance of SGA and EGA improves significantly. This can be seen through Figures 3a and 3h. In the triangle program the branch identifying an “equilateral triangle” was the most difficult to cover. But, as the mutation rate increased, this could be covered by SGA and EGA.

### 4.2 Rectangle Program

Figure 4 shows average number of generations vs. population and average percentage coverage vs. population curves for SGA, EGA, EGAFS and EGAFSM for the rectangle classification problem for five different mutation probabilities 0.001, 0.005, 0.01, 0.05 and 1.
For the mutation probabilities of 0.001, 0.005, and 0.01 the performance of EGAFSM is observably better (see Figures 4a through 4f). Although, 100% coverage is not achieved with EGAFSM in every run in the case of 0.001 for any of the population sizes experimented with (see Figure 4b), it could be achieved in the case of 0.005 for larger population sizes (≥ 40; see Figure 4d). The low average coverage with SGA and EGA can also be seen in Figures 4b, 4d and 4f. Full coverage is achieved for all population sizes in every run for EGAFS and EGAFSM for all mutation probabilities greater than or equal to 0.01. This is not the case with SGA which achieves this with a mutation probability greater than 0.05 (see Figure 4j). With all the strategies, as the mutation rate is increased, average percentage coverage improves and the average number of generations also
In the case of the mutation probability of 0.1, full coverage is achieved for all population sizes in every run of the experiment (Figure 4j). The average number of generations is also significantly reduced with this mutation probability, as can be seen in Figure 4i, as compared to the low mutation probabilities of 0.001 and 0.005 (see Figures 4a and 4c).

4.3 Date program

As we can see in Figure 5, in the Date program we observe similar behavior. Complete coverage could be obtained with mutation rate of 0.05 in all cases for all runs. Furthermore, with low mutation probability and small populations, the performance of EGAFSM outdoes the rest.

Figure 5. Date Program Results
4.4 An experiment with mutation

In order to observe test data generation behavior with mutation probability, we plotted the average number of generations with mutation probability and average percentage coverage with mutation probability. The graphs presented unusual results as can be seen in Figure 6 where the average number of generations is plotted on a logarithmic scale for population size 30, a moderately small population size. If we look at Figures 6a, 6c and 6e, we notice that the average numbers of generations reduce till about a mutation probability of 0.1 and then increase till about 0.7-0.8 and then reduce again. This suggests i) that the mutation probability may be the best to adopt, ii) high mutation probabilities, i.e., greater than 0.1 and less than 0.7-0.8 may lead to poor performance and iii) a high degree of randomness may be also helpful in achieving coverage in some cases. Although high mutation probability leads to hundred percent average coverage in the case of SGA, EGAFS and EGAFSM (see Figures 6b, 6d and 6f), in the case of EGA, the average number of generations increase and the average coverage reduces as the mutation probability becomes very high (see Figure 6d and 6f). It seems that increasing mutation probability and thereby increasing randomness has a detrimental effect.
5. IS THE STRATEGY USEFUL?
It is apparent from the results and discussions in the earlier sections, that EGAFSM, along with the proposed branch ordering strategy, gives good results on all benchmarks for mutation probability greater than or equal to 0.01. The question now is: How does this compare with the performance of a modified SGA (SGAP) that does not incorporate the proposed branch ordering, elitism, fitness scaling and memory. The results of an experiment with SGAP on the rectangle and date programs with the mutation probability of 0.01, along with the results for EGAFSM and SGA, are shown in Figures 7 through 10. In the case of SGAP, the branch ordering was based on the node numbering in the control flow graph presented in [3] for each of the benchmark programs.

It is clear that the performance of EGAFSM is observably better than both SGA and SGAP. Not only is the average number of generations for all population sizes for both the rectangle and date program significantly less (see Figure 7 and Figure 9), as full coverage could be achieved quickly in most cases, but EGAFSM also achieves one hundred percent coverage on all population sizes experimented with for the rectangle program (see Figure 8) and for population size of 30 and more for the date program (see Figure 10). SGAP does not achieve full coverage for any of the population sizes experimented with.

Although the performance of SGA was not always better than SGAP for small population sizes, but it may be noted that the very reason for considering EGAFSM was because of the problems associated with SGA implementation. It is also possible that the branch ordering derived from the node numbering in [3] was good enough to give better performance for SGAP than SGA.

6. CONCLUSION
In this paper we have described how the Path Prefix strategy can be used with Genetic Algorithms to generate test data for branch coverage. Considering the problems that were faced with the SGA implementation, elitism (EGA), elitism with fitness scaling (EGAFS) and elitism with fitness scaling and memory (EGAFSM) were considered along with the ordering strategy. In general it is observed that EGAFSM performs better than others both in terms of average number of generations and average percentage coverage. In all the GA variants, with mutation probability of 0.1, complete coverage could be achieved with all population sizes in fewer numbers of generations as compared to that with lower probabilities. However, with very high probabilities it was observed that there may be degradation in performance.

In order to ascertain whether there really was an improvement with EGAFSM, an experiment was carried out with a modified implementation of SGA (called SGAP here) that did not incorporate the branch ordering strategy, elitism, fitness scaling and memory. For a mutation probability of 0.01, the results of EGAFSM were compared with those obtained with SGAP. The results with EGAFSM were found to be comparably superior.

In an application of genetic algorithm for test data generation for branch coverage, it is important that we exploit the properties of evolution. Each time a branch is selected for coverage, if there are chromosomes in the current population that traverse the sibling then it may become easier to evolve a test case that covers the branch. It is this that we have tried to achieve with the path prefix strategy, elitism and memory.
As future work, we are comparing these results with the results obtained with implementations of other metaheuristic algorithms such as particle swarm optimization etc. and we are also testing for statistical significance.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES


