Effective Test Data Generation using Genetic Algorithms

Meenakshi Gupta, Lecturer, Deptt. of Computer Applications, MAIMT, Jagadhri, Haryana, India
Garima Gupta, Lecturer, Deptt. of Computer Applications, Hindu Girls College, Jagadhri, Haryana, India

Abstract
Software testing is an indispensible part of software development. It increases the confidence of programmer and user in the reliability and accuracy of software. However, it is a laborious and time-consuming task. Almost half of the software development resources spend on testing the software. Automatic software testing can substantially reduce the cost of development of software. Further exhaustive software testing is not feasible. Only the selective parts of the software are tested. Therefore design of a set of test cases is required in such a manner that it can find out as many faults as possible. We propose to improve software-testing efficiency with suitable optimization techniques. In this paper, the focus is on the use of genetic algorithms for generating the test data that can cover the most error-prone path; so that emphasis can be given on testing these paths firstly. Genetic algorithms are iterative techniques that apply simple operations repeatedly in the search for good solutions, or in this case, test data. By finding out the most error-prone path using this technique will help to reduce the software development cost and improve the testing efficiency.

Introduction
Software testing is the process of validation and verification of the software. Effective software testing contributes to the delivery of reliable and quality oriented software product, more satisfied users, lower maintenance cost. Whereas, ineffective testing leads to the opposite results; low quality products, unhappy users, increased maintenance costs. Hence, software testing is a necessary and important activity of software development process. It is the process of executing a program with the intent of finding errors. The importance of testing can be understood by the fact that “around 35% of the elapsed time and over 50% of the total cost are expending in testing programs”. Software is expected to work, meeting customer’s changing demands, first time and every time, consistently and predictably. Earlier software systems were used for back-office and non-critical operations of organizations. Now more and more critical applications are implemented globally. This increased expectation for error-free functioning of software has increased the demand for quality output from software vendors. Software Testing is the process used to help and identify the correctness, completeness, security, reliability & quality of developed software. It is one of the major phases in all the phases of Software Development Life Cycle. The potential cost savings from handling software errors within a development cycle, rather than the subsequent cycles, has been estimated at nearly 40 billion dollars by the National Institute of Standards and Technology. This figure emphasizes that current testing methods are often inadequate, and hence reduction of software bugs and errors is an important area of research with a substantial payoff [1]. Therefore reducing the efforts, time and ultimately the cost of software development had always been a challenge for both the software industry and academia. The automatic generation of test data using Genetic Algorithms (GA) has been studied by [3, 4, 5, 6]. Here the use of GA is explored to automatically generate test data that covers the most error-prone path. GA are commonly applied to search problems within AI. They maintain a population of structures that evolve according to rules of selection, mutation and reproduction. Translating these concepts to the problem of test data generation, the population is set of test data. Each element in the set (e.g. a group of data items used in one run of the program) is an individual [5]. The fitness of an individual corresponds to the coverage of an error-prone path of the program under test.

Genetic Algorithms: An Overview
Genetic Algorithms are computerized search and optimization algorithms based on the principle of natural genetics and natural selection. These are good for navigating very large search spaces looking for optimal combinations of things and solutions, which we might not find in a lifetime. These need design space to convert into genetic space. A genetic algorithm is a programming technique that mimics biological evolution as a problem-solving strategy. Evolution is under the influence of two fundamental processes; natural selection and...
recombination. The former determines which individual member of a population is selected, survives and reproduces, the latter ensures that the genes (or entire chromosome) will be mixed to form a new one [9]. Figure 1 depicts the structure of a simple GA.

- begin GA
- g:=0 ; generation counter
- initialize population P(g)
- evaluate population P(g) ; compute fitness value
- while not done do
  - g:=g+1
  - select P(g) from P(g-1)
  - crossover P(g)
  - mutate P(g)
  - evaluate P(g)
  - reinsert

Fig. 1 Structure of Genetic

The quality of individuals that constitute the current population P(g) is evaluated according to a scalar fitness function. Individuals that achieve a higher fitness are more likely to be selected as parents and generate offspring by means of recombination and mutation. Recombination promotes the exchange of genetic information among individual members of the population. The offspring are subject to mutations, which randomly modify a gene in order to create new variants. The current population is replaced by the newly generated group of offspring, which forms the new population P(g + 1) in the next generation. The evolutionary algorithm terminates either if a maximum number of generations elapses or a desired level of fitness level is reached [7, 8].

Steps for Automatic Test Data Generation

Test-data selection, and consequently generation, is all about locating test-data for a particular test criterion. T[4]est data generation for path testing consists of four basic steps [4]:
1) In this step, the source program is transferred to a graph that represents the control flow of the program.
2) Target path selection: In path testing, paths are extracted from the control flow graph, and some paths might be very meaningful and need to be selected as target path for testing.
3) Test case generation and execution: In this step, the algorithm automatically creates new test cases to execute new path and leads the control flow to the target path. Finally, a suitable test case that executes the target paths could be generated.
4) Test result evaluation: This step is to execute the selected path and to determine the test criteria is satisfied.

Genetic Algorithms and Test Data Generation

Genetic Algorithms are commonly used to search problems in the field of artificial intelligence. This section describes generation of test data using GA that achieve a certain level of coverage of the program. Our approach uses a weighted Control Flow Graph (CFG) technique. Path testing searches the program domain for suitable test cases that covers every possible path in the software under test. However, it is generally impossible to achieve this goal due to following reasons[3].
- A program may contain an infinite number of paths when the program has loops.
- The number of paths in a program is exponential to the number of branches in it and many of them may be unfeasible.
- The number of test cases is too large, since each path can be covered by several test cases.

Since it is impossible to cover all paths in software, the problem of path testing selects a subset of paths to execute and find test data to cover it. Here a program is viewed as control flow graph. It is a simple notation for the representation of control
The control flow of a program can be represented by a directed graph with a set of nodes and a set of edges \([3, 4, 6, 10]\). Each node represents a statement. The edges of the graph are then possible transfers of control flow between the nodes. A path is a finite sequence of nodes connected by edges. An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph an independent path must move along at least edge that has not been traversed before the path is defined.

**Setting of different GA Parameters** [3]
Input - CFG of code
Assign weights to edges of CFG – More weights are assigned to edges that are more error prone. Firstly weight is assigned to initial node of CFG. If the CFG contains large number of edges then large weight is assigned to first node, otherwise small weight such as 10 is assigned. Then on the basis of this initial node weights are assigned to other nodes. Incoming weight is divided and distributed to all the outgoing edges of the node. More weight is given to branches and loops and less weight is given to edges of sequential path. A ratio of 4:1 is used for this purpose. If there is only one outgoing edge from a particular node than the incoming weight is assigned to the outgoing edge. The CFG for factorial function is shown in figure 2.

```c
factorial (int num)
{
    int prod, m, i;
    prod=1;
    m=num;
    i=1;
    while (i<=m) {
        prod=prod*i;
        i=i+1;
    }
    return prod;
}
```

**Initialization**
An initial test set is generated randomly in the space of possible input values.

**Selection**
The selection of parents for reproduction is done according to a probability distribution based on the individual’s fitness values. The fitness value is calculated using the following function:

\[
F = \sum_{i=1}^{n} wi
\]

Here \(wi\) is weight assigned to \(i\)th edge on the path and \(n\) is initial population size.
After all the fitness values are calculated, the probability of selection \(pj\) for each path \(j\) is calculated, so that

\[
pj = \frac{Fj}{\sum Fj}
\]

where \(j=1\) to \(n\)
Then cumulative probability \(ck\) is calculated for each path \(k\) with equation:

\[
ck = \sum_{i=1}^{k} pj
\]

**Crossover**
Crossover probability (Cp) is decided. It is an adjustable parameter. For each parent selected, a random real number \(r\) is generated in the range \([0, 1]\); if \(r < Cp\) then select the parent for crossover. After that, the selected data is formatted randomly. Each pair of parents generates two new paths,
called offspring. For the problem in hand, one point crossover is suitable.

**Mutation**

Mutation probability (Mp) is decided. It is an adjustable parameter. To perform mutation, for each chromosome in the offspring and for each bit within the chromosome, generate a random real number r in the range [0, 1]; if r < Mp then mutate the bit.

These major components including the fitness function will evolve test data to better ones, trying to find a candidate that covers the target path. The crossover process tries to create better test data from fitter ones, while mutation introduces diversity into population, avoiding being stuck at local optima solutions. According to algorithm given in figure 1, the process is repeated using GA parameters until a suitable test data is generated. Various studies show that GA performs better from one generation to the next to cover selected paths. According to Aljabidi et al [2009], GA improves the search from one generation to the next, and performs better than random testing, where the search was absolute random and does not show improvement through the generations. Double crossover is more successful in path coverage. Also selecting parent for reproduction according to their fitness is more efficient than random selection and mutation rate is better adjusted with program at hand.

Sthamer [1995] says GA requires up to two orders of magnitude fewer tests than random testing and achieves 100% branch coverage. The advantage of GAs is that through the search and optimisation process, test sets are improved such that they are at or close to the input subdomain boundaries. According to C.C. Michael et al, test data generation using GA performs better compared to random test data generation.

The comparative results on small math programs with the goal of achieving condition decision-coverage are shown in Table 1. Genetic search outperformed random test data generation by a considerable margin in most of programs and always performed at least as well.

**Conclusion**

Automatic test data generation is not only a current research topic but also a critical problem in software testing. The methodology for the automatic test data generation tries to detect a set of test cases that lead to fulfil a given criterion, and thus results in the reduction of cost in software testing. In this paper, overview and possibilities of genetic algorithms to automatically generate test data are presented. The application of genetic algorithms to software test data generation differs from the optimization problem where a single goal is required. Here the optimization is finding the most error-prone path in the code under test. Automatic test data generation will build confidence in the desired properties of the software. Further research in this area will ultimately reduce the costs associated with software testing.

**References**


---

<table>
<thead>
<tr>
<th>Program</th>
<th>Random</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>53.3</td>
<td>66.7</td>
</tr>
<tr>
<td>Bubble sort 1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Bubble sort 2</td>
<td>44.4</td>
<td>44.4</td>
</tr>
<tr>
<td>Number of days</td>
<td>35.3</td>
<td>39.2</td>
</tr>
<tr>
<td>between two dates</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Euclidean GCD</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Computing the median</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Quadratic formula</td>
<td>91.7</td>
<td>100</td>
</tr>
<tr>
<td>Warshall’s algorithm</td>
<td>48.6</td>
<td>84.3</td>
</tr>
</tbody>
</table>

Table 1. Comparative results of test data generation using GA and random testing.


