

Using Genetic Algorithms to Generate Test Suites

Austin Barket and Randall Hudson

Department of Computer Science
The Pennsylvania State University at Harrisburg
Middletown, PA 17057
amb6470@psu.edu, rvh5220@psu.edu

Abstract. This paper applies techniques of genetic algorithms to the problem of finding test suites that cover every edge or predicate in a programs control flow graph. It differs in other applications of genetic algorithms to software testing in that the organisms are test suites rather than test cases. Experimental results show....

1 TODO

Should we, as suggested by El Ariss, replace all references to testing programs with testing functions?

In our test suite crossover function we still call adapt mutation probability. Should we talk about this in the paper, just leave it out of the paper and keep it in the program. Or just remove it from the program since it doesn't really make sense with what the algorithm became?

- Add any missing citations
- Decide if to use FloatBarrier or not
- Our Work section
 - Begin with short, broad overview, then begin algorithm section
 - Pick Main Loop algorithm
 - FindRanges algorithm
 - AdaptRanges alg
 - For each algorithm, discuss it briefly in words
- Experimental results section
 - Discuss the programs we tested on
 - Maybe give a theoretical running time for a random searcher to find inputs for HardCFG
 - Give results (Tables and graphs)
- Discussion Section
 -
- Conclusion Section
 - Concluding remarks
 - Future work
- Read over the copy and pasted sections from proposal to make sure it all makes sense in this new context.
- Add a little bit more to the survey section, specifically more modern stuff

2 Introduction

Software testing is a tedious and expensive process, costing as much as 50% of a projects budget [1]. One reason for this is the fact that the input domain for programs tends to be extremely large, often times infinite; this makes it impossible to test all possible inputs for a program [1]. Thus, testers spend a large portion of their time simply searching/designing good tests. One aspect of automatic testing research aims to alleviate this burden via the automatic generation of tests.

Automatic test generation can be characterized by the creation of some system that when fed information about a program will attempt to generate good tests for that program. In model based testing, one such method for automatic test generation, engineers must design a model of the system using the functional requirements for the system; the model then generates tests cases to test the functional requirements [2]. In rule based test generation, testers design a framework of rules, generally built around predicate statements; this framework then dictates the creations of test cases based on the predicate statements in the code [3]. These are two of the common methods used in automatic test generation. These methods generally require some upfront design costs before use.

Genetic algorithms represent a third, and perhaps more interesting, method of automatic test generation. Tests, or test suites, take the role of the organisms in this genetic algorithm while their coverage, e.g. branch, conditional, etc, determines their fitness [4]. This method represented an underexploited area of research and will be further examined in this paper.

3 Problem Definition

In high-level terms the problem succinctly presents itself as such: automatically generate good tests for a given program; but this leaves several open questions, such as “what information if any can be gotten from the program to help design the tests?” and “what constitutes a good test?” The field of structural testing will help answer both of these questions.

In any given program there are multiple paths in which flow of control can traverse; these can be represented by a control flow graph (CFG); predicates, i.e. conditional statements in the code, determine which path to take [1]. Structural testing encompasses the use of such information to design tests [1].

When evaluating a test’s coverage, the predicates, and the flow of control they affect, play an important role. Consider the following simple metric: A test suite should cover every possible path in a program at least once. Using the control flow graph one can easily evaluate the quality of tests against this metric. A good test suite then becomes one in which most if not all of the possible paths are taken at least once. Structural testing uses this metric, and similar metrics, as its primary evaluation method for tests.

The problem to solve can now be stated with more detail: automatically create a test suite that covers all, or most, paths in a programs control flow graph. It is the authors’ intention to solve this problem with a genetic algorithm.

To design a genetic algorithm that solves this problem it will need the following information: the input parameters for the program, the control flow graph, and the predicates affecting each path. The output would consist of a population of test cases or test suites, depending on design decisions, that should consist of good tests.

4 Related Work

Using genetic algorithms to generate tests represents a relatively new area of research, with the first published results dating from only 1992 [5]. Initially, genetic algorithms evolved tests that would ensure all predicates for a single path of execution were true[6]. Later, Watkins created an adaptive fitness function that would create tests for one particular branch of the control flow graph and then switch to a different branch once some tests were found; this approach allowed a single run of the genetic algorithm to attempt to evolve the tests needed for full branch coverage[6]. Watson’s results showed that a genetic algorithm, compared to random testing, required “an order of magnitude fewer tests to achieve path coverage”[6]

Up to this point, the fitness functions viewed a branch’s predicate as just a boolean function, that is true or false; this did not take full advantage of white box testing in which the exact details of each predicate can be known. Sthammer wished to use information about these predicates to create boundary value tests. He accomplished this by using a distance function that reported the distance between a test’s current values for a predicate and what values that predicate needed to be true. Sthammer’s showed that his technique preformed better then a random search.

Pargas et al proposed the used of a control-dependence graph instead of a control flow graph to guide the search of test cases. They implemented this technique in a parallelizable algorithm called GenerateData[8]. When tested on programs with control flow graphs of cyclomatic complexity of seven significant improvements over random search were reported[8].

The newest genetic algorithms attempt to make use of past information in some way. In Beuno and Jino’s research they used information about past runs of the genetic algorithm to influence the creation of the initial population used in a new run[6]. “The results with their six small test programs were promising.”[6] Berndt et al try to take advantage of past information in a different way. Their algorithm includes a “fossil record that records past organisms, allowing any current fitness calculations to be influence by past generation.”[6]

Those interested in a more detailed survey are encouraged to investigate the survey done by Aljahdali et al, on which this survey is partially based on [6].

TODO: add a few more recent papers

5 Our Work

Our approach focused on the idea of using test suites as an organism's chromosome rather than test cases. We also added some new evolutionary algorithm techniques that did not appear to be widely used in other literature for this problem. Additionally, we implemented several test programs of varying difficulty. Each of these items will now be discussed in turn.

As noted above, test suites compose the population and each of these consists of multiple test cases. The number of test cases for a given test suite was decided to be the sum of the number of edges and predicates in the target control flow graph; this would ensure every test suite would be able to cover all possible branches and predicates in the target program. Each test case contains an array of values that represent input parameters to the program being tested. For the purposes of this research, input parameters were restricted to the domain of 32-bit integers.

Since an organism's chromosome is now an entire test suite, one may think the simulation should try to evolve organisms with the maximum possible coverage, but this is not the case. Even if no single organism has full coverage, if the population as a whole covers everything, then there is no need to continue the simulation. In order to still get a single test suite with full coverage, as the simulation discovers test cases that offer new coverage, it records their inputs.

Many of the GAs from the literature for this problem are rather simplistic **TODO: add citations**. To improve upon them we implemented and tested many techniques, including several types of fitness scaling, several base fitness functions, several mutation operators for test suites, tournament selection, several types of local optimizations, several adaptive parameters et al Most of these were not fit enough to survive our culling. The subset that did survive includes local optimization for test cases, fitness sharing, and adaptive ranges. These are discussed in more detail in the algorithm sub-section.

Test programs are provided to the algorithm in the form of executable CFGs. A CFG represents linear blocks of instructions from the target program as nodes. The edges represent possible different directions the flow of control can follow when it reaches a predicate. When implementing multiple condition coverage, one could simply expand each predicate, turning it into one predicate for each possible outcome; this has its advantages but can make tracking branch versus MCC coverage harder. Our implementation uses edges solely for possible directions of control in the original target program. A separate set for predicates is kept, and each node (aka block) is responsible for tracking its predicates. This means that in our implementation, covering all edges equates to achieving 100% branch coverage and covering all predicates equates to achieving 100% multiple condition coverage.

5.1 Algorithm

At a high level the algorithm is based around a simple observation from the field of software testing. Programs, although having very large input domains,

typically only have a few equivalence classes of their parameters. Here an equivalence class refers to a range of values for a parameter such that when input to the program cause the same flow of control. e.g. A program with one input and the simple predicate $x < 0$, has two equivalence classes for the value of x $[\text{int_min}, -1]$ and $[0, \text{int_max}]$. As a result this program only needs two test cases for achieve full coverage.

Of course, unlike the problem above, most meaningful programs have several parameters and typically have complex relationships between input parameters that drive the control flow of the program. The algorithm proposed here finds much of its basis in a simplifying assumption of these complex relationships. Consider a program with three inputs x , y , and z . Let $R1$, $R2$, and $R3$ be ranges in the domains of these inputs. Suppose this program requires a test case of the following form to cover a specific path in the CFG $\{x \in R1, y \in R2, z \in R3\}$. The assumption made is that $\{x \in R1, y \in R1, z \in R1\}$, $\{x \in R2, y \in R2, z \in R2\}$, and $\{x \in R3, y \in R3, z \in R3\}$ will each cover different paths in the control flow graph.

The algorithm attempts to find a set of promising ranges using this assumption. Then applies a genetic algorithm to search the space of test cases resulting from various combinations of parameters from these ranges. Of course this is not perfect and it would not be difficult to construct a program that violates this assumption. However our results show that this method can be effective in finding the vast majority of coverage in test programs with a range of complexity.

The following section has been broken down into the subsections for each major aspect of the proposed algorithm. First, the main simulation loop is introduced to give context throughout the rest of the sections. The notion of input parameter ranges and the simulation's range set is then introduced, followed by a discussion of the population initialization. Fitness calculation and association with organisms is explored; then the tools used for test case generation, including test case crossover and mutation, test suite crossover and mutation, local optimization and range set adaptation are studied.

Main Simulation Loop The main simulation loop has several components that need to be addressed. First, it must keep track of how many generations it has been since no improvement was made. The simulation uses this variable as part of the ending predicate and in determining how often test suite crossover and range adaptation is preformed. **TODO: What else to say here**

```

run(maxGens, numCutPoints, Pm)
  currentGen  $\leftarrow$  0

  FindGoodRanges()
  BuildPopulation()

  do {
    updatePopulationsFitness()

    TestCaseCrossover(numCutPts, Pm)

    if currentGen is multiple of 10
      TryLocalOptimization()

    if no improvement for more than 20 generations or currentGen is multiple of 30
      AdaptRanges()
      TestSuiteCrossover(numCutPts, Pm)

    currentGen  $\leftarrow$  currentGen + 1
  } while ending criteria has not been met

```

Fig. 1: Algorithm for main simulation loop

Initializing the Range Set The algorithm maintains a set of promising input parameter ranges throughout the simulation. Any time a new test case needs to be generated the simulation can query this range set for good ranges; the simulation can then pick parameters from those ranges for the new test case.

A range includes a start value and an end value, as well as an array of input parameter buckets. The buckets are partitions of size 25 within the range, e.g. the range [0, 74] would contain the following array of buckets [[0, 24], [25, 49], [50, 74]]. These buckets contain counts of how many times that sub-range was used to provide some unique coverage for the population. Summing the counts in these buckets gives the usefulness for that range.

At the start of the simulation the FindGoodRanges() procedure is run to find promising ranges of input values and add them to the global range set. This procedure loops, starting with a range around zero and extending in both directions towards max and min integer, generating a single test case from each range. If a test case provides coverage that has not yet been seen during this initial procedure, its corresponding range is added to the global range set.

This procedure can be repeated using various range sizes, which in some cases enabled the algorithm to find additional coverage and ranges not found in previous iterations. However, this is an expensive procedure; thus, only two iterations were used, first with a range of size 5000, then with a range of size

2500. This provided a good set of initial seed ranges for the genetic algorithm to utilize, while not taking too long to complete.

```

FindGoodRanges()
  rangeSet ← Empty Range Set

  for ( size ∈ {5000, 2500} )
    posStart ← -size/2
    negStart ← size/2

    while posStart + size < max_int and negStart - size > min_int {
      posRange ← [posStart, posStart + size]
      negRange ← [negStart, negStart - size]

      posTC ← test case in posRange
      negTC ← test case in negRange

      if (posTC covers anything new)
        add posRange to rangeSet

      if (negTC covers anything new)
        add negRange to rangeSet

      posStart ← posStart + size
      negStart ← negStart - size
    }
  end-for

  return rangeSet

```

Fig. 2: Algorithm for finding good ranges

Initializing the Population During initialization each organism in the population uses only a single range from the simulation’s range set to generate all its test cases. In this way the algorithm starts by searching the space of test cases possible using only parameters from a single range. The genetic operators defined below then attempt to search the possible combinations of these ranges, focusing on the ranges and organisms that have been most fruitful in generating unique coverage thus far.

Fitness

Coverage Metadata The algorithm maintains three levels of coverage metadata for the purposes of fitness calculation. First, each test case is assigned coverage by

running it against the CFG. Each test suite then maintains counts of the number of times its constituent test cases covered each edge and predicate. Finally, the population maintains sums of the coverage counts across all of the test suites.

Fitness Sharing The algorithm assigns fitness for an organism based off both its coverage and the population's coverage; this is a form of fitness sharing. Each edge or predicate has a base score associated with it. The score for a particular edge or predicate is split amongst all the test suites that cover it. These scores are then split amongst all the test suites that cover them. The motivation for using fitness sharing is the following: if many organisms cover a particular edge, then it must be easy to reach and thus organisms should receive less fitness for covering it; however, if only one organism in the population covers an edge, then that organism should receive a higher fitness. Furthermore if a edge is covered by multiple organisms, but only a handful of times population wide, the organisms that cover that edge should receive more fitness than organism who cover an edge that has been covered a very large number of times population wide.

Test Case Generation and Search The main test case search occurs by crossing over and mutating test cases within organisms, as well as applying local optimization heuristics that manipulate or replace an organism's test cases. As mentioned above the algorithm begins by thoroughly searching the space of possible test cases composed of parameters from singular ranges of the range set. After many generations of no improvements using test case crossover and local optimization, the algorithm performs two key perturbation procedures. First the range set adapts itself based on information learned regarding the fruitfulness of its constituent ranges. Then test suite crossover and mutation are applied to mix up the available input parameters inside of test suites.

Crossover Test suites and test cases each have their own crossover operator. Both operators use normal k-point crossover and fitness proportional selection. However, the test case operator simply selects one test suite and then selects test cases at random from that test suite. The simulation attempts test case crossover several times before giving up; this is to increase the chance of finding a good parameter combinations, assuming the range search has narrowed in on good ranges.

Mutation is also executed during this step. The simulation uses point mutations for both, each with probability P_m . For test suites mutation involves simply creating a new test case, for test cases it involves changing a parameter to a new value picked using a normal distribution around the parameters original value.

After test suite crossover, the algorithm always replaces both parents, while test case crossover only replaces a test case if it offers new coverage for a test suite.

Even though these two operators are similar they each serve a different purpose. Test suite crossover causes a greater change in the partaking organisms by adding and removing test cases. Since, each test case was created with certain

ranges, this also has the effect of mixing up the ranges in a test suite. Test case crossover, on the other hand, makes much smaller changes to a test suite, by mixing up the parameters between two test cases. The hope is that test suite crossover will help the test suite get the ranges it needs to find new coverage and then test case crossover will be able to combine those ranges in the correct manner.

```

TestCaseCrossover(cutPts,  $P_m$ )
  Proportionally select a test suite, ts, from the population

  for  $i = 0$  to 100
    From ts select two random test cases, tc1 and tc2

    child1, child2  $\leftarrow$  crossover(tc1, tc2, cutPts)
    child1.mutate( $P_m$ )
    child2.mutate( $P_m$ )

    if child1 is covering anything not covered by ts
      ts.addTestCase(child1)
      break
    else if child2 is covering anything not covered by ts
      ts.addTestCase(child2)
      break
  end-for

```

Fig. 3: Algorithm for test case crossover

Local Optimization Two simple local optimization techniques are used, but the structure of both is very similar. The LocalOptFromParameters function will take a test case and, beginning with a small neighbor size, will begin looking at other test cases in the neighborhood of the supplied one. The LocalOptFromZero function tries to pull the test case's parameters closer to zero. This was added because values in the neighborhood of zero, such as 1, -1, or 0, can drastically alter the flow of the program being tested, but because the search space was so large, the simulation was unable to find these values without this. Both functions explore each neighborhood several times before expanding the neighborhood. If at any point, a test case is found that covers something not in the population then it is added to the organism by replacing a duplicate test case.

```

TryLocalOptimization()
  org ← P.Select()

  Randomly select one of the two local opt functions and try it on a
  duplicate test case from org, store results in tc

  if tc is not null replace a duplicate test case in org with tc

```

Fig. 4: Algorithm for selection of a local optimization

```

LocalOptFromParameters(testCase)
  neighborhoodSize ← 5
  for i = 1 to 500
    temp ← testCase
    for j = 1 to 150
      for each parameter, p in temp
        set p to a random value in the neighborhood of p
      end-for
      if temp covers anything not covered in the population return temp
    end-for
    neighborhoodSize ← neighborhoodSize + 5
  end-for

  if no improvement was ever found then return null

```

Fig. 5: Algorithm for local optimization from existing parameters

```

LocalOptFromZero(testCase)
  neighborhoodSize ← 5
  for i = 1 to 1000
    temp ← testCase
    for j = 1 to 150
      for each parameter, p in temp
        95% of the time set p to a random value in the neighbor of zero
        5% of the time set p to p/2
      end-for
      if temp covers anything not covered in the population return temp
    end-for
    neighborhoodSize ← neighborhoodSize + 5
  end-for

  if no improvement was ever found then return null

```

Fig. 6: Algorithm for local optimization around zero

Range Set Adaptation The initial search for good ranges will probably fail to find all useful ranges; it may also think some ranges are useful which actually are not. Range adaptation aims to give the simulation the ability to remove non-useful ranges, as well as focus in on the most fruitful ranges, and find new ones that may have been missed. As mentioned above this procedure is only run after several generations of no improvement using the other operators.

First, the range set deletes any ranges below one standard deviation usefulness. All ranges above one standard deviation usefulness are split into two equal sized ranges, with the input parameter buckets copied accordingly to the new ranges. The two ranges immediately above and below the original range are also added to the range set. These new adjacent ranges have the same size as the original range and their usefulness values are set to half that of the original range in order to prevent them being completely ignored. Finally, the range set adds a new random range into its pool to avoid missing ranges not found during initialization or a previous range adaptation.

```

AdaptRanges()
  for each range in rangeSet
    if (range.usefulness <  $\mu - \sigma$ )
      remove range from the range set

    if (range.usefulness >  $\mu + \sigma$ )
      addAdjacentRanges(range)
      splitRangeIntoTwo(range)
  end-for

  add new random range to the RangeSet

```

Fig. 7: Algorithm for adapting ranges

Stopping Criteria The algorithm has three potential stopping criteria.

1. Achieving 100% branch and multiple condition coverage.
2. 500 generations without any improvement of branch or multiple condition coverage
3. 10000 generations have passed.

In conclusion the proposed algorithm seeks to generate a test suite with maximal coverage through the use of a simplifying assumption of the relationships between input parameters and a genetic algorithm. First, the algorithm finds a set of promising ranges that give unique coverage when used as the sole source of parameter values. Then a genetic algorithm, with the help of local optimization and range set adaptation, searches the space of possible test cases stemming

from the most promising combinations of these ranges to build a test suite with maximal branch and multiple condition coverage. The results of our tests are promising, however further inspection uncovered that the majority of the test cases that made it into the final test suite originated from the initial population and local optimization, and the genetic search did not work as well as we hoped.

6 Experimental Results

The algorithm was tested against 5 control flow graphs of varying complexity, these include a simple too high or too low number guessing program, the well known triangle problem, and three larger custom built CFGs. Our testing consisted of two steps. First, we analyzed the effect of varying the parameters of population size, mutation probability, and number of cut points on some of the implemented graphs. Based on the results of these initial tests we set up a number of final tests to perform. These will be described in detail below.

6.1 Test Programs

HiLo CFG This graph models a very simple program with three integer input; num1, num2, and guess. An internal variable target is computed as the product between num1 and num2. The guess parameter is compared to the target to determining if the user's guess was correct, too high, too low, not even trying (less than both num1 and num2), or equal to 0. The resulting CFG has 10 blocks, 14 edges, and 14 predicates.

Triangle CFG The triangle problem is a very common introductory problem in the field of software testing. The program takes three inputs, each representing the length of a side of a triangle. The program then compares the sides to each other using a number of predicates to determine if the sides form an equilateral, isosceles, scalene, or violate the triangle inequality. The resulting CFG has 13 blocks, 12 edges, and 21 predicates.

Hard CFG In an attempt to give the algorithm a more formidable test, this graph was custom designed. The idea was to have some more complex relationships between the parameters dictate the shifts in control flow. Thus in addition to conditional statements involving the traditional comparison operators between parameters, this graph utilizes the mean, standard deviation, and the sum of the parameters in its predicates. This graph takes the form of a perfect binary tree and is generally symmetric in nature as far as the types of predicates in corresponding blocks of the left and right subtrees. The hard CFG has 10 parameters, 31 blocks, 78 edges, and 150 predicates.

Med Hard 1 and Med Hard 2 CFGs Initial testing seemed to show that Hard CFG was a little too challenging for the algorithm. In addition it exceeded the scope of the problems algorithms such as these are typically evaluated against. Thus the Hard CFG was split at the root into two subtrees, which became Mad Hard 1 and Med Hard 2. The number of parameters was pruned down to 4, naturally each subtree has 39 edges, and due to how the cut was performed Med Hard one has 74 predicates while Med Hard 2 has 79 predicates.

6.2 Parameter Tuning

In hopes of finding an optimal combination of parameters for the GA, systematic tests were run to generate a number of 3D graphs representing slices of the 4D space of {number of cut points, mutation prob., pop. size, number of generations}. To this end the Hilo, Triangle, Med Hard 1, and Med Hard 2 each were evaluated with population sizes ranging from 5 to 100, number of cut points ranging from 1 to 4, and mutation probability ranging from 0 to 1. Population sizes were stepped by 5, cut points by 1, and mutation probability by 0.02. Thirty runs for each combination of parameters were averaged to get the reported results.

The results on all graphs showed that essentially none of the parameters made a substantial impact on the ability of the algorithm to find coverage. Coverage ratios were generally constant across the board for each graph, regardless of what parameters were used. The only exception being that if the population size was less than the number of ranges found in the initial range search, coverage suffered by as much as 10%. For HiLo and Triangle the algorithm maxed out coverage in less than 100 generations. For the med hard graphs the number of generations varied by no more than 400 generations, and in a mostly random manner making it hard to claim any particular combination of parameters was better than another. This seemed to indicate that the initial range search, population generation, and local optimization functions account for the majority of the coverage obtained by the algorithm.

6.3 Results

Since no particular set of parameters seemed to prevail of others, we chose to run our final tests somewhat arbitrarily with a population size of 25, 0.5 mutation probability, and 2 cut points. In order to further investigate the efficacy of the various operators, three categories of final tests were performed on each of the CFGs. First, we evaluated each CFG against the entire algorithm as described above. Then, to determine if in fact the genetic operators were capable of finding good test cases on their own, we evaluated each CFG against the algorithm without any calls to either local optimization heuristic. Finally, to determine the effectiveness of the local optimization alone, we cut out all calls to test case and test suite crossover and mutation, as well as range adaption to test only the initial range search, population initialization, and local optimization operations.

The results...

CFG	Branch Coverage	MCC Coverage	Avg. Generations	Avg. seconds/run
HiLo	0.857143	0.785714	753.4	1.88864
Triangle	0.916667	0.904762	578.92	1.60156
Medium-Hard-One	0.846154	0.816901	1257.96	6.51491
Medium-Hard-Two	0.666667	0.658228	1073.38	6.22098
Hard	0.564103	0.46	1208.94	17.9978

Fig. 8: Final Results, No Local Optimization

7 Discussion

8 Conclusion

8.1 Future Work

References

1. G. Meyers, *The Arts of Software Testing*. Hoboken, New Jersey: John Wiley & Sons Inc. 2nd edition, 2004.
2. M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco: Morgan-Kaufmann. 2007.
3. K.-H. Chang, J. H. C. II, W. H. Carlisle and D. B. Brown *A framework for intelligent test data generation*. Journal of Intelligent and Robotic Systems vol. 5, no. 2, pp. 147-165, April 1992.
4. J. Wegener, H. Sthammer, B. Jones and D. Eyres, *Testing real-time systems using genetic algorithms*. Software Quality Journal. pp. 127-135, 1997.
5. X. S, E. C, S. C, L. G. A, K. S and K. K, *Applications of Genetic Algorithms to Software Testing*. in Internation Conference on Software Engineering and its Applications, Toulouse, 1992.**TODO: fix this citation**
6. S. Aljahdali, Taif, A. Ghiduk and M. El-Telbany, *The limitations of genetic algorithms in software testing*. in Computer Systems and Applications, Hammamet, 2010.
7. B. F. Jones, H. H. Sthammer and D. Eyres, *Automatic structural testing using genetic algorithms*. Software Engineering Journal, pp. 299-306, September 1996.
8. R. P. Pargas, M. J. Harrold and R. R. Peck, *Test-Data Generation Using Genetic Algorithms*. Journal of Software Testing, Verification, and Reliability, 1999.