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 the algorithm attains high coverage on the tested programs; however, further analysis brings into question their significance.

1 Introduction

Software testing is a tedious and expensive process, costing as much as 50% of a projects budget [3]. 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 [3]. 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 [4]. 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 [5]. 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 [6]. This method represented an underexploited area of research and will be further examined in this paper.

2 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 test suite that should reach a high coverage metric.

3 Related Work

Using genetic algorithms to generate tests represents a relatively new area of research, with the first published results dating from only 1992 [7]. Initially, genetic algorithms evolved tests that would ensure all predicates for a single path of execution were true[8]. 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[8]. Watson's results showed that a genetic algorithm, compared to random testing, required "an order of magnitude fewer tests to achieve path coverage"[8]

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[10]. When tested on programs with control flow graphs of cyclomatic complexity of seven significant improvements over random search were reported[10].

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[8]. "The results with their six small test programs were promising."[8] 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."[8]

More recent work has been done by Gordon Fraser, Andrea Arcui, and others in their development of the evolutionary test suite generator EvoSuite [1]. This is some of the earliest research using test suites instead of test cases for the organism's chromosome. Test cases are also unique in that they are composed of Java statements rather than the program's input parameters. Both test cases and test suites can change size during the simulation. EvoSuite has proven to be a powerful method for automatic test generation [2].

4 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 organisms 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.

In an attempt to improve upon GAs from the literature, 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 forms of adaptive mutation,

and others. 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.

4.1 Algorithm

At a high level the algorithm is based around a simple observation from the field of software testing. Programs typically only make use of a small subset of their input domains, thus it is wasteful to search the entire space of possible test cases with equal probability. 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. With MCC, test cases in the ranges that satisfy the constituent conditions of a compound predicate will have differing coverage. Thus these ranges can be discovered easily. Once discovered these ranges can be combined to saisfy the more complicated compound conditions.

The following section has been broken down into the subsections for each major aspect of the proposed algorithm. First, the main simulation loop is provided 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.

Simulation Algorithm

```
currentGen ← 0

FindGoodRanges()
BuildPopulation()

do {
   UpdatePopulationsFitness()

   Proportionally select a test suite, ts, from the population
   Use ts for test case crossover and mutation to get tc
   If tc is not null add it to ts by replacing a duplicate test case

   if currentGen is multiple of 10
      TryLocalOptimization()

  if no improvement for ≥ 20 generations and currentGen is multiple of 100
     AdaptRanges()

  if no improvement for ≥ 20 generations and currentGen is multiple of 30
     Proportionally select two test suites from the population
     Do crossover and mutation on these to get children
     Replace both parents with the children
  fi

  currentGen ← 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 > 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. 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.

Crossover, Mutation, and Replacement 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 only selects one test suite and then selects test cases at random from that test suite to crossover. The simulation attempts test case crossover several times before giving up; this is to increase the chance of finding good parameter combinations, assuming the range search has narrowed in on good ranges.

Mutation and replacement immediately follow both forms of crossover. Both the test suite and test case mutation operators use the same probability P_m. Test suite mutation involves creating new test cases out of the most useful ranges in the range set. These are added to the child test suites only if they offer new coverage and in such a way to ensure no previous coverage is lost. After test suite crossover and mutation, the algorithm always replaces both parents. For test cases, mutation involves changing a parameter to a new value selected using a normal distribution around the parameter's original value. Child test cases only find their way into the parent test suite if they offer new coverage.

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 swapping a significant number of entire test cases between the parents. Since, each test case was created with certain ranges, this also has the effect of mixing up the ranges in a test suite. Since this is a disruptive operation, the algorithm performs test suite crossover only after several generations of no improvement in coverage. Test case crossover, on the other hand, makes much smaller changes to a test suite, by mixing up the parameters between two test cases. Thus the ranges of parameters available within the organism do not change, but the coverage of the suite may improve as a result. This was designed to be the main genetic operator for test case search, and thus is run every generation. The hope is that test suite crossover will help the suite get the needed ranges and then test case crossover will be able to combine those ranges into more useful test cases.

```
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 ← 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 neighborhood size, will begin looking at other test cases in the neighborhood of that test case. 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 help. Both functions explore each neighborhood several times, then expand the neighborhood to try again if no new coverage could be found. The TryLocalOptimization procedure called from the main simulation loop chooses, with equal probability, one of these heuristics to apply. If at any point during either procedure a test case is found that offers new coverage to the simulation, it is added to the organism by replacing a duplicate test case.

```
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. 4: 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. 5: 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. This procedure is only run after many 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 < μ − σ)
         remove range from the range set

      if (range.usefulness > μ + σ)
         addAdjacentRanges(range)
         splitRangeIntoTwo(range)
   end-for

   add new random range to the RangeSet
```

Fig. 6: 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.

Summary The proposed genetic algorithm seeks to generate a test suite with maximal coverage by utilizing a simplifying assumption of the relationships between input parameters. 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.

5 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.

5.1 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.

HardLeft and HardRight CFG The hard graph is quite a bit larger than benchmark tests used by previous researchers. In addition, initial speculation of results on Hard CFG, seemed to indicate that the right subtree was more difficult to cover than the right subtree. To analyze this more closely, the Hard CFG was split at the root into two subtrees, which became HardLeft and HardRight. The number of parameters was pruned down to 4, each subtree has 39 edges, and due to how the cut was performed HardLeft has 74 predicates while HardRight has 79 predicates.

5.2 Parameter Tuning

In hopes of finding an optimal combination of parameters for the GA, systematic tests were run on all the graphs just discussed. The details of these tests can be seen in Appendix A.

The results on all graphs showed that 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.

5.3 Results

Setup Since no particular set of parameters seemed to prevail over 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, four categories of final tests were performed on each of the CFGs. First, we evaluated each CFG against the entire algorithm as described in section 4.1. Then, to determine the effectiveness of the local optimization alone, we cut out all calls to test case and test suite crossover and mutation. To determine the extent the genetic operators are capable of finding good test cases, we evaluated each CFG against the algorithm without any calls to either local optimization heuristics. Finally, each CFG was run against a completely random search for comparison. The random search was given the same amount of time as the average time taken by a run of the entire algorithm on each graph. Each test was repeated 50 times and the results shown in the following tables represent the averages obtained.

Description of Tables Tables 1, 2, and 3 report the results from running the entire algorithm, the algorithm without genetic operators, and the algorithm without local optimization respectively. The top halves of the tables report the average branch and multiple condition coverage ratios, as well as the number of generations taken and the average run time of the simulation. In the lower portion of the tables, the reader will find the average size, in number of test cases, of the test suites that resulted from the runs in table, as well as the percentage of test cases in the final suite that came from each of the algorithm's five test case sources. Table 4 contains the coverage ratios achieved by a completely random searcher given the same amount of time taken by the algorithm to get the results in Table 1.

	HiLo	Triangle	Hard-Left	Hard-Right	Hard
Branch Coverage	100%	100 %	92 %	92%	95%
MCC Coverage	100%	96 %	87 %	90%	85%
Generations	46	521	1027	1300	1824
Run Time	1s	6s	26s	34s	95s
Suite Size	9	15	30	34	60
% Init Pop.	58%	75%	35%	28%	21%
% TestSuite Op	0%	0%	0%	0%	0%
% TestCase Op	2%	5%	10%	11%	10%
% L.O. (param)	9%	9%	32%	14%	20%
% L.O. (zero)	31%	11%	23%	46%	49%

Table 1: Coverage achieved on normal run

	HiLo	Triangle	Hard-Left	Hard-Right	Hard
Branch Coverage	100%	100 %	90 %	92%	88%
MCC Coverage	100%	96 %	83 %	89%	81%
Generations	52	530	1140	1430	1788
Run Time	1s	5s	27s	33s	86s
Suite Size	9	15	30	34	60
% Init Pop.	57%	75%	35%	28%	21%
% TestSuite Op	0%	0%	0%	0%	0%
% TestCase Op	0%	0%	0%	0%	0%
% L.O. (param)	12%	11%	37%	22%	25%
% L.O. (zero)	31%	14%	28%	51%	53%

Table 2: Without Crossover and Mutation

	HiLo	Triangle	Hard-Left	Hard-Right	Hard
Branch Coverage	86%	92%	77 %	70%	54%
MCC Coverage	79%	90%	72 %	67%	43%
Generations	729	579	1290	1096	1255
Run Time	2s	2s	7s	7s	19s
Suite Size	7	14	24	19	28
% Init Pop.	76%	81%	43%	47%	46%
% TestSuite Op	1%	0%	18%	1%	15%
% TestCase Op	23%	19%	40%	52%	39%
% L.O. (param)	0%	0%	0%	0%	0%
% L.O. (zero)	0%	0%	0%	0%	0%

Table 3: No Local Optimization

	HiLo	Triangle	Hard-Left	Hard-Right	Hard
Branch Coverage	78%	59 %	13 %	13%	13%
MCC Coverage	64%	71 %	6 %	6%	6%
Run Time	1s	6s	26s	34s	95s

Table 4: Random Search

6 Discussion

In table 1 we see that the overall branch and multiple condition coverage ratios achieved by the algorithm are quite high. Deeper analysis of the missing predicate for the triangle graph showed that it is actually unreachable, meaning the algorithm does in fact achieve 100% coverage of this graph. Despite these exciting results achieved when running the algorithm as a whole, the results of the algorithm without crossover and mutation, as well as the GA alone without local optimization bring into question their significance as well as the usefulness of the implemented CFGs as software testing benchmarks.

As suspected from the parameter tuning tests, the algorithm achieves very similar coverage of the implemented CFGs with or without the crossover and mutation operators. From table 1 we see that when running the entire algorithm, anywhere from 75 to 90% of the test cases that make up the final test suites for the hard graphs were either present in the initial population or were generated as the result of one of the local optimization heuristics. Comparison of table 1 with table 2 shows that cutting out mutation and crossover all together does not significantly change the coverage achieved by the algorithm. In one sense this means that the FindGoodRanges procedure can be extremely effective in discovering ranges of parameters that traverse different paths of the CFG. Furthermore, LocalOptFromZero and LocalOptFromParameters perform quite well in finding test cases that navigate the delicate and complex relationships between the sums, means, standard deviations, and ranges of the parameters. In another sense, this may be an indication that the Hard CFG was not as challenging as expected and thus does not form a good benchmark for the algorithm.

In the case of the hard graphs, the algorithm only failed to cover a very specific type of compound predicate. Namely, those which required two or more parameters to be exactly equal to each other, while also having the sum, the mean, and the standard deviations of the parameters falling within particular ranges. One could imagine adding a simple local optimization or test case generation heuristic that would attempt combinations of having several parameters equal to each other. With this, its likely that the algorithm could be extended to achieve 100% coverage on all of these test programs. However, this approach of simply adding additional heuristics to get the coverage we desire does not address the underlying problem. Perhaps the main problem with this algorithm's ability to search for test cases stems from the use of test suites as organisms. This design decision made it hard to use the fitness to drive the search for new test cases. By assigning fitness to entire test suites instead of test cases, it is very difficult to tell if you're getting closer or further away from new coverage. We attempted to address this problem by adding the FindGoodRanges and localOptFromZero procedures; however, these are targeted towards the implemented CFGs and would likely not be useful in general. In addition to the bad fitness function, the algorithm suffers from poor design of test suite mutation and range adaptation.

The results for the percent contributions of test cases from test suite mutation are also concerning and elucidate a clear weak point of the algorithm. Tables 1

and 3 show that this operator almost never succeeded in generating useful test cases. As mentioned briefly in section 4.1, this operator attempts to combine parameters selected from the most useful ranges in the range set into test cases. Selection of ranges is done proportional to the usefulness of the ranges in the set. Therefore, the fact that it fails to generate good test cases seems to indicate that the method used to assign usefulness values to ranges, as well as adapt the contents of the range set based on usefulness, are likely flawed and need to be revisited.

Although the initial range search and local optimizations are able to achieve most of the coverage on their own, the entire algorithm still outperformed the runs without crossover and mutation by as much as 7% branch coverage and 4% predicate coverage on the hard CFGs. Thus, the genetic operators are not completely disposable. Even without local optimization, the GA still achieves respectable levels of coverage on each of the graphs in only a fourth of the time taken by the algorithm with the local optimization included. This striking time difference demonstrates that although the local opts perform quite well and can find a significant number of test cases and coverage missed by the GA alone, they do not scale well and would not be near as valuable on larger more complex programs.

7 Conclusion

7.1 Future Work

Appendix A

The algorithm takes three input parameters, population size, mutation probability, and number of cut points. In order to find a good set of parameter, tests were run in which we kept just one variable constant while trying to optimize the other two. For example, a test would keep population size constant at 10 while letting the mutation probability range from 0.00 to 1.00 and letting the number of cut points range from 1 to 4. A "step size" is used to avoid trying too many values. We first test the mutation probability at 0.00 and then for the next test we step it up to its next value. Each parameter, with its starting value, ending value, and step size can be seen in the following table.

	Start	End	Step
Population	5	100	5
Mutation	0	1	0.05
Cut Points	1	4	1

References

1. Gordon Fraser, Andrea Arcuri, *Whole Test Suite Generation* IEEE. 2011.
2. Gordon Fraser, Matt Statts, Phil McMinn, Andrea Arcuri, Frank Padberg *Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study.* ACM Transactions on Software Engineering and Methodology (TOSEM). 2014
3. G. Meyers, *The Arts of Software Testing.* Hoboken, New Jersey: John Wiley & Sons Inc. 2nd edition, 2004.
4. M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* San Francisco: Morgan-Kaufmann. 2007.
5. 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.
6. J. Wegener, H. Sthammer, B. Jones and D. Eyres, *Testing real-time systems using genetic algorithms.* Software Quality Journal. pp. 127-135, 1997.
7. 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**
8. S. Aljahdali, Taif, A. Ghiduk and M. El-Telbany, *The limitations of genetic algorithms in software testing.* in Computer Systems and Applications, Hammamet, 2010.
9. B. F. Jones, H. H. Sthammer and D. Eyres, *Automatic structural testing using genetic algorithms.* Software Engineering Journal, pp. 299-306, September 1996.
10. R. P. Pargas, M. J. Harrold and R. R. Peck, *Test-Data Generation Using Genetic Algorithms.* Journal of Software Testing, Verifcation, and Reliability, 1999.