*Journal of Software Testing, Verification and Reliability. 1999, to appear, Wiley copyright*

# Test-Data Generation Using Genetic Algorithms

Roy P. Pargas
Department of Computer Science
Clemson University
Clemson, SC USA 29634-1906
pargas@cs.clemson.edu

Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
Columbus, OH USA 43210-1277
harrold@cis.ohio-state.edu

Robert R. Peck
Department of Computer Science
Clemson University
Clemson, SC USA 29634-1906
rpeck@cs.clemson.edu

**Abstract**

This paper presents a technique that uses a genetic algorithm for automatic test-data generation. A *genetic algorithm* is a heuristic that mimics the evolution of natural species in searching for the optimal solution to a problem. In the test-data generation application, the solution sought by the genetic algorithm is test data that causes execution of a given statement, branch, path, or definition-use pair in the program under test. The test-data-generation technique was implemented in a tool called `TGen` in which parallel processing was used to improve the performance of the search. To experiment with `TGen`, a random test-data generator, called `Random`, was also implemented. Both `TGen` and `Random` were used to experiment with the generation of test-data for statement and branch coverage of six programs.

Keywords. Software testing, test-data generation, genetic algorithms.

## 1 Introduction

Software testing is an expensive component of software development and maintenance [11]. A particularly labor-intensive component of this testing process is the generation of test data to satisfy testing requirements. Given a testing requirement, such as an input that will cause execution of a particular statement, test-data generation techniques attempt to find a program input that will satisfy the testing requirement. The automation of test-data generation is an important step in reducing the cost of software development and maintenance.

A number of test-data generation techniques have been automated. *Random* test-data generators (e.g., [18, 20, 22]) select random inputs for the test data from some distribution. *Structural* or *path*-oriented test-data generators typically use the program's control-flow graph, select a particular path, and use a technique such as symbolic evaluation to generate test data for that path (e.g., [1, 3, 6, 15, 19]). *Goal-oriented* test-data generators select inputs to execute the selected goal, such as a statement, irrespective of the path taken (e.g., [7, 16]). *Intelligent* test-data generators often rely on sophisticated analyses of the code, to guide the search for new test data (e.g., [2, 17, 21]).

The weaknesses of these techniques, however, have inhibited their widespread use for generation of test data. A random test-data generator may create many test data; however, because information about the test requirement is not incorporated into the generation process, the test-data generator may fail to find test data to satisfy the requirement. A path oriented test-data generator first identifies a path for which test data is to be generated; however, because the path may be infeasible, the test-data generator may fail to find an input that will traverse the path. An intelligent approach may, if the analysis is accurate, generate new test data quickly; however, the analysis required for this success over a variety of programs may be quite complex and may require great insight on the part of the designer of the test-data generator to anticipate many different programming situations.

This paper presents a goal-oriented technique for automatic test-data generation that uses a genetic algorithm, which is guided by the control dependencies in the program, to search for test data to satisfy test requirements. The genetic algorithm conducts its search by constructing new test data from previously generated test data that are evaluated as good candidates. The algorithm evaluates the candidate test data, and hence guides the *direction* of the search, using the program's control-dependence graph. To reduce the execution time of a potentially lengthy search, the prototype implementation uses multiple processors, and balances the load automatically to prevent processors from getting locked in time-consuming loops.

The main benefit of the approach presented is that, because it combines the salient features of random, goal-oriented, and intelligent test-data generators, it can generate test data quickly, but with focus and direction. New test cases are generated by applying simple operations on existing test cases that are judged to have good potential to satisfy the test requirements. Because the operations are simple, the new test cases are generated quickly and efficiently. Because the existing test data are judged as having good potential, the search is focused instead of being random. The success of this approach, therefore, depends heavily on the way in which the existing test data are measured — the empirical results suggest that the use of the control-dependence graph provides an effective way to perform this measurement.

Another benefit of the approach presented is that it has the potential to scale for large software systems. The approach can handle test-data generation for programs with multiple procedures. The evaluation of test data can be weighted in a manner that encourages generation of test data that cover test requirements embedded in nested procedure calls. The approach is also designed for parallel execution. Test requirements can be distributed among as many processors as the user has available. The approach automatically load balances and cannot continue infinitely searching for test data for an infeasible path: a processor that has unsuccessfully met a test requirement within a predetermined time period is instructed to abort the search and to start a new search with a different test requirement. These features let the user conduct *simultaneously* independent searches for test data to satisfy different test requirements.

Finally, the technique will be useful during regression testing where the regression test suite can be used. The regression test suite may be used as part of the initial set of test data. Additionally, because this approach can be focused on specific test requirements, only those test requirements that are part of *new* source code need to be targeted.

In the next section, background on genetic algorithms and control-dependence graphs is given. In the following section, the genetic algorithm for test-data generation is presented. Then, the implementation of the algorithm and the results of an empirical study using the implementation are discussed. Next, the technique presented is compared to related work. Finally, conclusions and future work are given.

## 2 Background

This section gives an overview of genetic algorithms and control-dependence graphs.

### 2.1 Genetic Algorithms

A *genetic algorithm* is an optimization heuristic that mimics natural processes, such as selection and mutation in natural evolution, to evolve solutions to problems whose solution spaces are impractical for traditional search techniques, such as *branch-and-bound*, or optimization techniques, such as *linear programming.* Since first described by Holland [14], genetic algorithms have been applied to a variety of learning and optimization problems.[1]

A genetic algorithm typically begins with a random population of solutions (*chromosomes*) and, through a *recombination* process and *mutation* operations, gradually evolves the population toward an optimal solution. Obtaining an optimal solution is not guaranteed — the challenge is to design the process to maximize the probability of obtaining such a solution. The first step is the selection of the solutions in the current population that will serve as parents in the next generation of solutions. This selection requires that the solutions be evaluated for their *fitness* as parents: solutions that are closer to an optimal solution are judged higher, or *more fit*, than others.

After solutions have been evaluated, several are selected in a manner that is biased toward the solutions with higher fitness values. The reason for the bias is that a good solution is assumed to be composed of good components (*genes*). Selecting such solutions as parents increases the chances that their offspring will inherit these genes and will be at least as fit. Although the selection is biased toward the better solutions, the worst members of the population still have a chance of being selected as parents — even a poor solution may have a few good genes that may benefit the population. After selection, the parents are recombined or mutated to produce offspring. The resulting solutions form the new population, and the cycle is repeated.

Evaluation, selection, recombination, and mutation may be performed many times in a genetic algorithm; thus, they must be as simple as possible. Selection, recombination, and mutation are generic operations in any genetic algorithm and have been well studied in the literature. Evaluation, however, is problem dependent and relates directly to the structure of the solutions. The major design decisions in the development of a genetic algorithm involve the structure of the solutions and the method of evaluation. Other design decisions include size of the population, the frequency of recombination relative to mutation, the dynamic nature of the frequencies, and the type of population replacement. Many of these decisions are made after one or more small trial runs of an initial genetic algorithm.

### 2.2 Control Dependence Graphs

Control dependence for a program is defined in terms of the program's control-flow graph and the postdominance relation that exists among the nodes in the control-flow graph [8]. In a *control-flow graph*, nodes represent statements, and edges represent the flow of control between statements — an edge (X, Y) in a control-flow graph means that program control can flow from X to Y. To facilitate analysis, a control-flow graph is augmented with unique *entry* and *exit* nodes. Given such a control-flow graph, and nodes W and

[1] Goldberg [12] gives an introduction to the theory behind genetic algorithms; Davis provides a clear guide in his handbook [5] and gives a comparative study between genetic algorithms and another optimization heuristic called *simulated annealing* [4].

```
     Program Example
        integer i, j, k
1.      read i, j, k
2.      if (i < j)
3.         if (j < k)
4.            i = k;
           else
5.            k = i;
           endif
        endif
6.      print i, j, k
     end Example
```

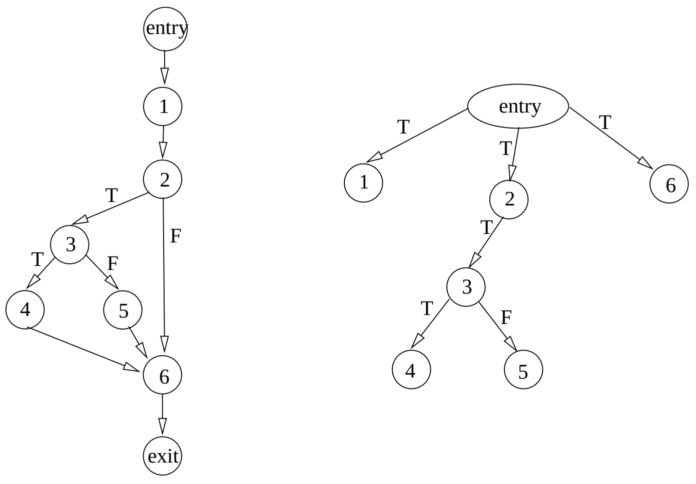


Figure 1: Program `Example` on the left, its control-flow graph (CFG) in the center, and its control-dependence graph (CDG) on the right.

V in that graph, W is *postdominated* by V if every directed path from W to the exit (not including W or exit) contains V. For statements (nodes) X and Y in a control-flow graph, Y is *control dependent* on X if and only if (1) there exists a directed path P from X to Y with all Z in P (excluding X and Y) postdominated by Y and (2) X is not postdominated by Y. In a *control-dependence graph*, nodes represent statements, and edges represent the control dependencies between statements — an edge (X,Y) in a control-dependence graph means that Y is control dependent on X. For example, Figure 1 depicts program `Example` on the left, its control-flow graph in the center, and its control-dependence graph on the right; nodes in the graphs are labeled with the statement number of the corresponding statement. In the control-flow graph, node 6 postdominates all nodes except itself and exit, nodes 3, 4, and 5 postdominate no nodes in the graph, node 2 postdominates nodes 1 and entry, and node 1 postdominates entry. Node 4 is control dependent on 3T, node 5 is control dependent on 3F, and nodes 1, 2, and 6 are control dependent on entering the program.

An acyclic path in the control-dependence graph from the root of the graph to a node in the graph contains a set of predicates that must be satisfied by an input that causes the statement associated with the node to be executed; such a path is called a *control-dependence predicate path.* For example, in Figure 1, the set of predicates {entryT, 2T, 3T} constitutes a control-dependence predicate path for statement 4. Unstructured transfers of control, such as **goto**, **continue**, and **break**, can cause the occurrence of more than one control-dependence predicate path for statements following the transfers. However, the number of control-dependence predicate paths is generally small.

## 3 Genetic Algorithm for Test-Data Generation

Figure 2 shows `GenerateData`, the algorithm for automatic test data generation. `GenerateData` uses a genetic algorithm to search for test cases that satisfy desired testing requirements. A solution (*chromosome*) is a set of test data (i.e., a list of input values). The algorithm evaluates test data by executing the program with the test data as input, and recording the predicates in the program that execute with that test data. This list of predicates is compared with the set of predicates found on the control-dependence predicate paths for the node representing the current test requirement that is the target of the search. A test data's

```
algorithm   GenerateData
input       Program : instrumented version of a program to be tested
            CDG : control-dependence graph for Program
            Initial : initial population of test cases
            TestReq : list of (test requirements, mark)
output      Final : set of test cases for P
            TestReq : list of (test requirements, mark)
declare     CDGPaths : a set of predicate paths in CDG
            Scoreboard : record of satisfied test requirements
            CurPopulation, NewPopulation : set of test cases
            Target : test requirement for which a test case is to be generated
            MaxAttempts() : function that returns true when maximum number of attempts
               for a single Target has been exceeded,
            OutOfTime() : function that returns true when the time limit is exceeded,
              and false otherwise
  begin
/* Step 1: Initialization and set up*/
[1]     Create CDGPaths
[2]     Create and Initialize Scoreboard
[3]     Generate CurPopulation
/* Step 2: Generate test cases */
[4]     while ((some (r,unmarked) ∈ TestReq) and not OutOfTime()) do
[5]         Select unmarked Target from TestReq
[6]         while (Target not marked and not MaxAttempts() do
[7]             Compute fitness values of CurPopulation using CDGPaths
[8]             Sort CurPopulation according to fitness
[9]             Select parents of NewPopulation
[10]            Generate NewPopulation from selected members of CurPopulation
[11]            Execute Program on each member of NewPopulation
[12]            Update Scoreboard and mark TestReq to reflect those test requirements that are satisfied
[13]        endwhile
[14]    endwhile
/* Step 3: Clean up and return */
[15]    Final = test cases that satisfy TestReq
[16]    return (Final, TestReq)
  end
```

Figure 2: Algorithm to automatically generate test cases for a program.

fitness evaluation depends on the number of predicates that it has in common with the predicates on a control-dependence predicate path of the target: a solution that covers the greatest number of predicates is given the highest fitness evaluation. `GenerateData` biases its selection of parents for the next generation of test cases using the fitness evaluation, and uses recombination and mutation to generate a new population of test cases.

The use of a genetic algorithm for `GenerateData` is atypical because the algorithm does not search for a single optimal solution. Rather, `GenerateData` has multiple goals — each test requirement that is the target of a search is a goal. Thus, throughout the search, the goal of `GenerateData` changes as test requirements are satisfied. Although the evaluation function remains unchanged, each set of test data must be re-evaluated with each new test requirement that is to be covered. Thus, the fitness evaluation of a set of test data changes as the test requirement changes: test data that may be valuable when attempting to cover one test requirement may be a poor candidate when attempting to cover another test requirement.

`GenerateData` takes the following as input: *Program*, an instrumented version of a program to be tested; *CDG*, the control-dependence graph for the program; *Initial*, an initial set of test data, possibly empty; and *TestReq*, a set of test requirements to be satisfied. Using local variables, *CDGPaths*, *Scoreboard*, *CurPopulation*, *NewPopulation*, and *Target*, and functions, *MaxAttempts()* and *OutOfTime()*, `GenerateData` outputs *Final*, a set of test data, and *TestReq*, the list of satisfied (marked) test requirements.

## 3.1 Step 1 of `GenerateData`

The first step of `GenerateData` (lines 1-3) is an initialization and preprocessing step, which includes computation of the *CDGPaths*, initialization of *Scoreboard*, and initialization of *CurPopulation.* First, for each statement in *Program*, `GenerateData` generates *CDGPaths*, which are the acyclic paths that contain the predicate nodes that lie between that statement node and the entry node of the *CDG*. Next, `GenerateData` creates *Scoreboard*, which records the satisfied *TestReq.* For statement or branch coverage, for example, *Scoreboard* may be a bit vector in which a bit (initially **false**) is set whenever the corresponding statement or branch is satisfied by a test data. As a second example, if the execution frequency of statements or branches is desired, *Scoreboard* may be an integer vector in which each element, initially zero, is incremented by one whenever the corresponding statement or branch is satisfied by a test case. Finally, `GenerateData` generates *CurPopulation* by first adding *Initial* to *CurPopulation.* If additional test cases are required to provide greater diversity, the algorithm randomly generates test cases subject to constraints imposed by the user, and adds them to *CurPopulation.* Diversity in the population gives `GenerateData` a wider variety of test cases from which to select; this diversity in population typically translates into better performance for genetic algorithms [12].

## 3.2 Step 2 of `GenerateData`

The second step of `GenerateData` (lines 4-14) is the outer **while** loop, which generates the test cases. To account for the possibility of infeasible test requirements, such as statements and branches,[2], the **while** loop iterates until either all test requirements are *marked* or *OutOfTime()* returns **true**. Although reaching the time limit in the search for a test case does not prove that unmarked requirements are infeasible, it does suggest this possibility. After execution of `GenerateData`, therefore, all unmarked testing requirements must be manually inspected. A test requirement is marked if a test case is generated that covers the test requirement or the test requirement has somehow been identified as infeasible.

The inner **while** loop (lines 6-13) focuses on a single unmarked *Target* that is selected from the unmarked test requirements in *TestReq.* The **while** loop iterates until either the *Target* is satisfied by a test case or *MaxAttempts()* returns **true**. `GenerateData` evaluates each test case in *CurPopulation* relative to *Target* (line 7) using *CDGPaths*, and computes a fitness value for the test case: if $test_i$ contains more predicates in *CDGPaths* for *Target* than $test_j$, then `GenerateData` assigns $test_i$ a higher fitness value than $test_j$. The algorithm then sorts *CurPopulation* based on these fitness values (line 8).

The use of the control-dependence graph in the evaluation of the fitness of test cases bears further discussion. The "intelligence" of the genetic algorithm in seeking to cover *Target* is derived from the use of the control-dependence graph. The premise is that developing a test case to cover a given *Target* is better

[2]A statement (branch, path) is *infeasible* if no input exists that will cause execution of that statement (branch, path).

done with test cases that come close to covering *Target* than with random test cases or with test cases that are nowhere near *Target.* For `GenerateData`, a test case "comes close" to satisfying a `Target` if it has a high number of predicates in common with a *CDGPath* of the *Target.* A test case that has a high number of predicates in common with a *CDGPath* of the *Target* satisfies many of the conditions required for execution of the *Target.* If a test case comes close to covering a given *Target*, it is likely that some portions of the test case are "good" because they caused execution near the *Target.* It is also likely that, by recombining good test cases, the algorithm has a better chance than a random approach of producing a test case that covers *Target.* It is believed that this use of the control-dependence graph is the reason that `GenerateData` significantly outperforms `Random` in three of the six programs in the case study; this study is discussed in Section 4.2. However, further evaluations that compare `GenerateData` with other test-data generation strategies are needed to evaluate the effectiveness of the approach.

After computing the fitness of each test case in *CurPopulation*, `GenerateData` selects test cases from *CurPopulation* that will be parents of *NewPopulation* (line 9); `GenerateData` biases this selection towards those test cases in *CurPopulation* with the highest fitness values. A test case with a high fitness value has a high probability of being selected, and may be selected multiple times; a test case with a low fitness value has a small, possibly zero, probability of being selected, and may never be selected. `GenerateData` assigns probabilities to the tests using the following algorithm. Given test case $t_i$ and its fitness value $f_i$, the probability assigned to $t_i$ is $p_i = f_i / \sum_{j=1}^{n} f_j$ where $n$ is the number of test cases in the population. Note that $\sum_{i=1}^{n} p_i = 1$. `GenerateData` then generates a random number $r$ $(0 \leq r \leq 1)$ and selects test case $t_j$ such that $\sum_{i=1}^{j} f_i \leq r$ and $\sum_{i=1}^{j+1} f_i > r$. The probability of selecting an arbitrary test case $t_i$ is exactly $p_i$.

The algorithm repeats this selection process until there are enough parents to generate *NewPopulation.* Two parents are required for recombination, which produces two offspring. One parent is required for mutation, which produces one offspring. Because the number of parents required for each operation equals the number of offspring produced, the total number of parents required is the size of *NewPopulation.*

After the selection process, `GenerateData` generates the *NewPopulation* by pairing the selected parents for recombination or considering them individually for mutation (line 10). Recombination, the first method for generating new test cases, takes two parent test cases and produces two children; it is the primary method of producing new test cases. The specific recombination method used in this study is *one-point crossover* [12]. Let $(a_0, a_1, \ldots a_k)$ and $(b_0, b_1, \ldots b_k)$ represent the $k+1$ input values of two selected test cases, *GenerateData* selects a random integer between 1 and $k$, inclusive. If the integer generated is $m$ $(1 \leq m \leq k)$, the two children produced are $(a_0, a_1, \ldots, a_{m-1}, b_m, b_{m+1}, \ldots, b_k)$ and $(b_0, b_1, \ldots, b_{m-1}, a_m, a_{m+1}, \ldots, a_k)$. These children are added to the new population. In the initial implementation of `GenerateData`, which is described in the next section, one-point crossover is used for recombination.

Mutation, the second method of generating new test cases, lets new *chromosomes* be introduced into the population; this is necessary to prevent stagnation, or freezing, of the population. Without mutation, the values of test cases in *NewPopulation* are limited to those values generated during *SetUp.* Mutation takes one of the selected test cases, $(a_0, a_1, \ldots a_k)$ generates a number $m$ $(0 \leq m \leq k)$ and replaces the value of input variable $m$, (i.e., $a_m$), with a randomly generated value. This mutation process creates a new, single, child test case that `GenerateData` adds to *NewPopulation.*

The relative frequency with which mutation is used is usually lower than that of recombination. In the current implementation, which is described in the next section, frequencies of 10% and 90% were used for

Table 1: Test Cases and Statement Traces for Example of Figure 1

| Test Case | Input Data $i, j, k$ | Statement Trace |
|---|---|---|
| $t1$ | 1, 6, 9 | 1, 2, 3, 4, 6 |
| $t2$ | 0, 1, 4 | 1, 2, 3, 4, 6 |
| $t3$ | 5, 0, 1 | 1, 2, 6 |
| $t4$ | 2, 2, 3 | 1, 2, 6 |

mutation and recombination, respectively. Both one-point crossover and mutation are simple operations, characteristic of genetic algorithms. The strategy is to employ fast and inexpensive operations, focused on covering the *Target*. Because the operations are fast and simple, they can be performed many times, increasing the chances that the *Target* will indeed be covered.

In the final step of the inner **while** loop, `GenerateData` executes *Program* on each member of *NewPopulation* (line 11), and updates *Scoreboard* accordingly. If at least one test case in *CurPopulation* satisfies *Target*, the algorithm marks *Target* in *TestReq*, the inner **while** loop terminates, and the algorithm attempts to find a new *Target*. Otherwise, *NewPopulation* is assigned to *CurPopulation*, and the algorithm makes another attempt to satisfy *Target*. If `GenerateData` fails to find a test case to satisfy *Target* before *MaxAttempts()*, the algorithm temporarily abandons *Target*, and selects the next test requirement in *TestReq* as the new *Target*. The entire search continues until either all test requirements are met or the time limit is exceeded, at which time the outer **while** loop terminates.

## 3.3 Step 3 of `GenerateData`

In this last step, the algorithm assigns the union of test cases that satisfied *TestReq* to *Final* (line 15), and returns *Final* and *TestReq* (line 16).

## 3.4 Example

To illustrate the operation of `GenerateData`, consider program `Example` and its CDG, which are shown in Figure 1. Suppose at some point during processing using `GenerateData`, *CurPopulation* contains the four test cases shown in Table 1. Inspection of the statement traces for the four test cases reveals that *CurPopulation* covers all statements except statement 5. Thus, `GenerateData` selects statement 5, whose *CDGpath* is {entryT, 2T, 3F}, for its search. In the **while** loop, the algorithm first computes the fitness values for each of the four test cases in *CurPopulation*. The fitness values of test cases $t1$ through $t4$ are $\{2, 2, 0, 0\}$, respectively (entry is ignored as a predicate because it is on all paths): $t1$ and $t2$ have two predicates (i.e., 2, 3) in common with *Target* and $t3$ and $t4$ share no predicates with *Target*. The fitness values of $t_1$ and $t_2$ ought to be higher because the paths associated with these test cases have more predicates in common with *Target* than the other two test cases. Thus, these test cases stand a better chance of contributing as parents to the next generation of test cases than $t_3$ and $t_4$.

Using the technique that was described for assigning probabilities, the algorithm assigns probabilities of {0.5, 0.5, 0.0, 0.0}, respectively, to the four test cases. Next, `GenerateData` selects the parents of *NewPopulation*. Suppose that the algorithm selects the following test cases in order $\{(1, 6, 9), (0, 1, 4), (0, 1, 4), (0, 1, 4)\}$; recall that a test case may be selected multiple times. Then, recombining the first two parents using one-point crossover, yields $t_5$(1,6,4) and $t_6$(0,1,9), and mutating the last two selected test cases yields $t_7$(0,6,4) and $t_8$(5,1,4). *NewPopulation*, therefore, is $\{t_5, t_6, t_7, t_8\}$, two of which ($t_5$ and $t_7$) satisfy the *Target*.

### 3.5 Analysis

Next, the major components that contribute to the complexity of `GenerateData` and the time complexity of a random approach are discussed. The following variables are used in the analyses: $n$, the number of statements in the original program; $P$, the population size; $e$, the execution time of the instrumented program; $k$, the depth of nesting of conditionals and loops in the source code; $r$, the number of testing requirements, $m$, the maximum attempts a process is allowed to generate test cases for a *Target*, and the length $L$, in bytes, of a test.

The time complexity of `GenerateData` is dominated by the initialization and set up (lines 1-3) and the loop that generates tests (lines 4-14). The initialization and set up consists of generation of the *CDGPaths*, which requires the control-dependence graph for the program ($O(n^2)$), generation of the *Scoreboard* ($O(n)$), and generation of the initial population *CurPopulation* ($O(P)$). Thus, this first step has time complexity $O(max(n^2, P))$. The loop that generates tests evaluates each test in *CurPopulation* for fitness, generates *NewPopulation*, and updates *Scoreboard*, all of which have $O(P)$ time complexity. The loop also sorts *CurPopulation* ($O(P \, log \, P)$) and executes the instrumented program on each test ($O(P \times e)$). Thus, each iteration of the inner loop has $O(max(P \times e, P \, log \, P))$. The number of times the inner loop is executed ranges from one to a maximum of $m$. Thus, the total execution time of the inner loop has complexity $O(m \times max(P \times e, P \, log \, P))$.

The outer loop (lines 4-14) executes until all test requirements are marked or until the time limit has been exceeded. This means that the outer loop can iterate zero or more times; the exact number varies from run to run. The time limit is set by the user and serves as an upper bound on the execution time of Step 2.

Unlike `GenerateData`, a random approach requires no set up or sorting of the population. Thus, the execution time of `Random` is at least $P \times e$ because it takes at least one execution of the instrumented program to determine whether all test requirements have been met. As with `GenerateData`, the user may set an upper bound on the execution time.

The space complexity of `GenerateData` depends on the major structures that it uses. *CDGPaths* is $O(n \times k)$ space; because $k$ may be considered constant (rarely greater than seven), this structure grows as $O(n)$. *CurPopulation, NewPopluation* is $O(P \times L)$; *Scoreboard* is $O(r)$. For statement coverage, $r$ is the number of statements, for branch coverage, $r$ is the number of branches. Thus, for statement and branch coverage, *Scoreboard* grows as $O(n)$. For path coverage, $r$ is the number of paths to be covered. Thus, for path coverage, the growth of *Scoreboard* depends on the number of paths generated.

## 4 Case Study

To determine the potential effectiveness of `GenerateData` over a random approach, a prototype of `GenerateData` in C that provides both statement and branch coverage was implemented; the discussion describes test-data generation only for statement coverage. With this prototype, an experiment involving a small set of C programs was performed. This section describes the implementation and experiments.

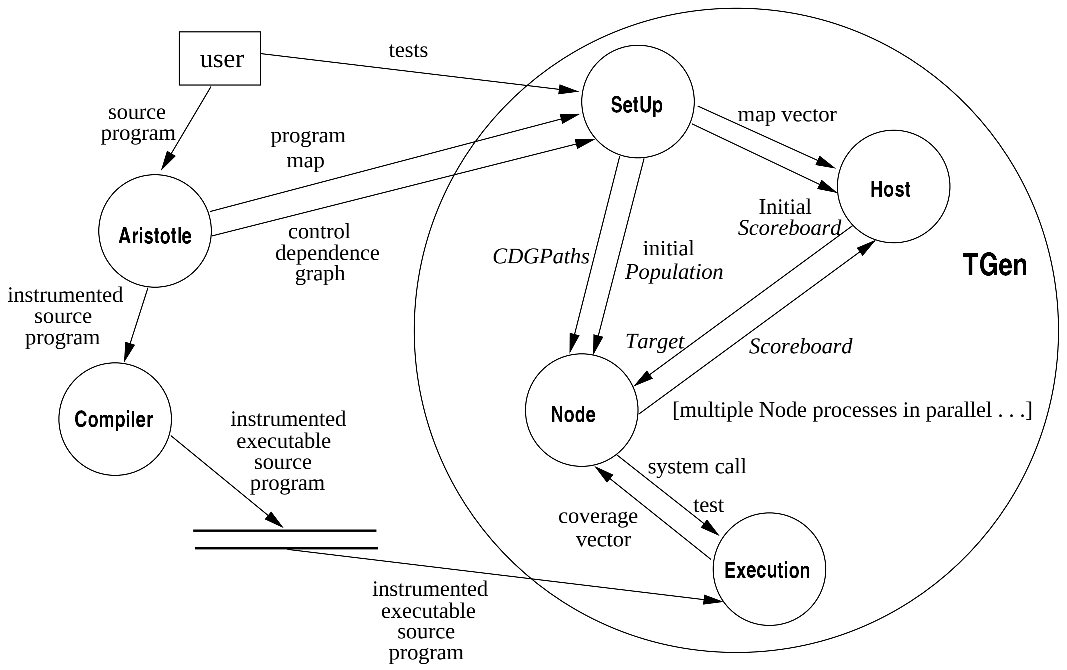


Figure 3: Dataflow diagram of the implementation of `TGen`.

## 4.1 Prototype Implementation

Figure 3 gives a dataflow diagram of the architecture of the implementation of `GenerateData`, called `TGen`. `TGen`-specific processes are shown inside the large circle in the figure. `TGen`, the prototype implementation of `GenerateData` makes use of the `Aristotle` analysis system [13] to generate a program map, a control-dependence graph, and an instrumented version of the program; the instrumented program is compiled, and the executable instrumented version of the program is stored in the file system.

`TGen` contains four major types of components: `SetUp`, which initializes the system; `Host`, which manages the dispatching of test-data generation jobs to `Node` components; `Node`, which implements our `TGen` algorithm to generate new tests; and `Execution`, which executes the program with the test cases generated by the associated `Node`. A particular implementation of `TGen` contains a single `SetUp` and `Host` component, but may have multiple `Nodes`, each executing on one processor.

The `SetUp` component of `TGen` inputs program map information, and generates the initial *Scoreboard* and a map vector that charts the locations and relationships among variables and predicates in the program. The `SetUp` component also inputs the control-dependence graph for the program, and computes the *CDGPaths* for statement nodes in the program. Finally, if the user provides an initial set of test cases, `SetUp` reads the test cases and passes them as the initial *Population* to each `Node` component.

The `Host` component of `TGen` acts as a manager for the test generation. This component inputs the map vector and the initial *Scoreboard*, and identifies and outputs one or more *Targets* to be satisfied. A `Node` component inputs the initial *Population* and the *CDGPaths* from `SetUp`, and a *Target* from the *Host*. A `Node` component works towards covering the *Target* that it receives from the *Host*. A `Node` performs its task by executing `TGen`, creating and evolving populations, and constantly striving to cover the *Target*. This version of `TGen` can generate test cases for statement or branch coverage. The `Node` returns its results to the `Host` if either the *Target* is covered or the maximum number of attempts allowed is exceeded. The `Node` outputs its results in the form of a *Scoreboard* that summarizes its attempts at satisfying a *Target*. The `Host` collects and merges the *Scoreboards* it gets from the `Node` processes until either the *Targets* are covered, or the time limit has been exceeded. If more *Targets* remain, `Nodes` receive additional *Targets*, and the cycle repeats.

The `Node` process issues an operating system call to cause the `Execution` to begin executing the instrumented program. Communication between the `Node` and the executing program accomplished using UNIX named pipes. The `Node` process passes the system call and a test case to the `Execution`, which loads the instrumented executable source program, executes this program, and returns information regarding which source program statements were covered.

`TGen` achieves parallelism by running multiple `Node` processes simultaneously on a network of Sun workstations running under either SunOS or Solaris operating systems. The user determines the number of workstations to be used in the system; this number is limited only by the number of workstations available on the local network. Each `Node` process runs on a separate workstation; a software package called Parallel Virtual Machines (PVM) [10] provides the communication and synchronization software tools required to allow `Host` and `Nodes` to work harmoniously. The parallel execution of `TGen` is a convenience and is not essential. The advantage of parallelism is the reduction in execution time, which may be a factor equal to the number of workstations used, allowing the user to run multiple runs in batch mode when network usage is low. However, `TGen`, as depicted in Figure 3, can just as easily be implemented on a single processor.

Although `TGen` currently handles statement and branch coverage, with minor modifications, it can also provide path and definition-use coverage. For path coverage, the *Target* is a specific path through the predicates of the program. Thus, to facilitate path coverage, the *Scoreboard* in `TGen` is altered so that each entry in it represents a set of statements rather than a single statement. Each entry in *Scoreboard* now represents one of these sets of statements, and the new objective of `TGEN` is to target statement sets rather than individual statements. The approach used for definition-use coverage is similar to the path-coverage approach.

`TGen` can also be extended to handle automatic generation of test data for programs with multiple procedures. Suppose that procedure A calls procedure B, that procedure B calls procedure C, and that statements in procedure C are to be covered. `TGen` proceeds as in the original algorithm with a slight modification in the evaluation function. If during test-case evaluation for fitness, the algorithm finds that the test case covers a call-site statement, it increases that test case's evaluation, over that obtained using the control-dependence graph, by a predetermined value multiplied by the number of marked statements it covers. This change in the method of evaluation of a test case causes those test cases that execute the call site from A to B to have a higher evaluation than those that do not. Furthermore, test cases that execute call sites from B to C are evaluated even higher. Higher evaluations for test cases translate into greater probability of selection. Hence, the population of test cases should tend to contain test cases that execute into B and C, letting `TGen` work toward those statements of interest in C.

## 4.2 Empirical Results

To determine the potential effectiveness of `TGen` over a random approach, a case study was performed on six C programs; Table 2 gives information about these programs. For each subject program, the table lists the number of lines of code, the cyclomatic complexity, and a short description of the program. For each

Table 2: Subject programs for our study.

| Programs | Lines of Code | Cyclomatic Complexity | Description of Program |
|---|---|---|---|
| Bub.c | 32 | 4 | Given an array of integers, bubble sorts the array |
| Find.c | 66 | 5 | Given array A[],and index F, places all elements less than or equal to A[F] to the left of A[F], and all elements that are greater to or equal to A[F] to the right of A[F] |
| Mid.c | 21 | 4 | Given three integers, determines the middle value |
| Bisect.c | 36 | 3 | Given an epsilon and X, computes sqrt(X) to within epsilon using bisection method |
| Fourballs.c | 82 | 7 | Given four integers representing the weights of balls, determines the weights of the balls relative to each other |
| Tritype.c | 61 | 7 | Given three integers, representing possible lengths of a triangle's sides determine the type of triangle (if any) |

Table 3: Results of `TGen` and `Random` runs for programs `Bisect.c`, `Fourballs.c`, and `Tritype.c`.

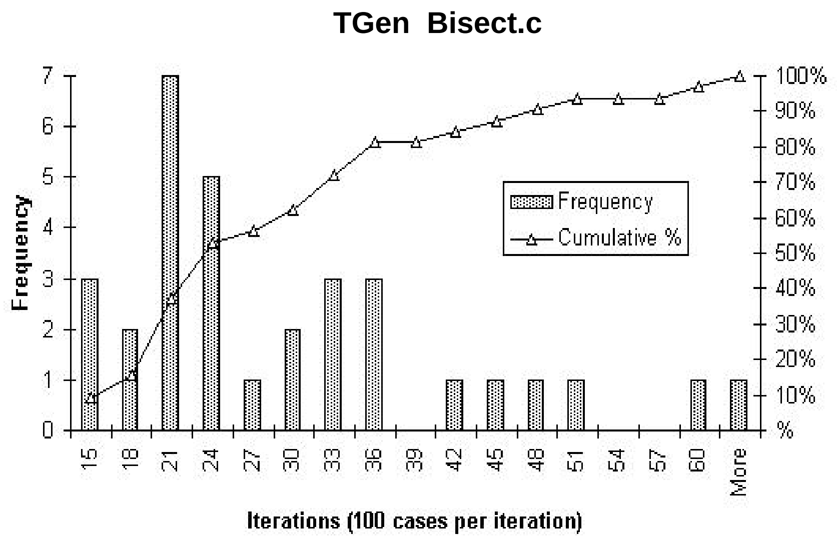
| Program | Generator | #Runs | Mean | Median | Mode | StdDev | Min | Max | Range |
|---|---|---|---|---|---|---|---|---|---|
| `Bisect.c` | TGen | 32 | 29 | 24 | 20 | 13 | 13 | 70 | 57 |
| | Random | 32 | 40 | 41 | 40 | 8 | 23 | 62 | 39 |
| `Fourballs.c` | TGen | 32 | 95 | 87 | 71 | 27 | 52 | 168 | 116 |
| | Random | 32 | 159 | 154 | 153 | 26 | 110 | 229 | 119 |
| `Tritype.c` statement | TGen | 32 | 145 | 134 | 49 | 97 | 21 | 392 | 371 |
| | Random | 32 | 1259 | 202 | n/a | 1141 | 24 | 3707 | 3683 |
| `Tritype.c` branch | TGen | 32 | 132 | 93 | 46 | 109 | 21 | 436 | 415 |
| | Random | 32 | 1100 | 691 | n/a | 1204 | 40 | 4881 | 4841 |

subject program, 32 `TGen` runs and 32 `Random` runs were performed. `Tritype.c` was used to generate test cases for statement coverage and branch coverage; the remaining programs were used to generate test cases for statement coverage.

The results of the experiments on three of the subject programs, `Bub.c`, `Find.c`, `Mid.c`, were virtually identical for both `TGen` and `Random` in that statement coverage was achieved immediately. The code did not involve nested conditionals, the predicates of the conditionals and loops were easily satisfied, and most of the code in the programs is straight line. `TGen`, achieved 100% statement coverage in the initial population. `Random` achieved similar results.

The results of the experiments on the other three programs, `Bisect.c`, `Fourballs.c`, and `Tritype.c`, however, were very different; Table 3 summarizes these results. `TGen` achieved 100% statement coverage of `Bisect.c` after iterating an average of 29 times; this is equivalent to generating 29 *NewPopulations*. `Random` achieved 100% statement coverage of `Bisect.c` after iterating an average of 40 times. (To produce a fair comparison, `Random` was designed to randomly generate sets of 100 test cases in one iteration, 100 being the size of the population of `TGen`.) The median and mode of `Random` are also higher than those of `TGen`.

Figure 4 summarizes graphically the results of the experiment with `Bisect` — `TGen` in on the top and `Random` is on the bottom. In each graph, the horizontal axis gives the number of iterations required by a run. The left vertical axis gives the frequency that such a run occurred. Thus, the sum of the heights of the vertical bars equals 32. The right vertical axis gives a cumulative frequency of the percentage of the 32 runs. Thus, three of the total 32 runs of `TGen` required 15 iterations to achieve 100% coverage of `bisect.c`. Over 50% (reading from the right vertical axis) of the 32 runs required 24 or fewer iterations.



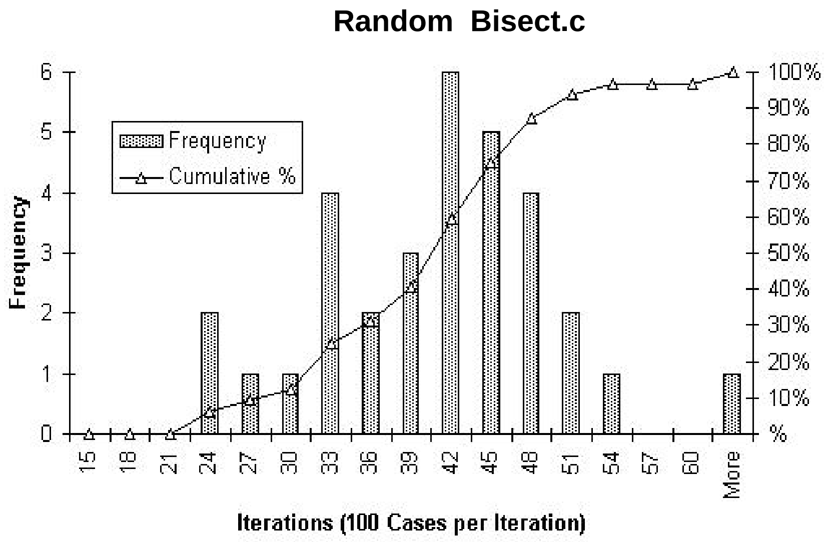
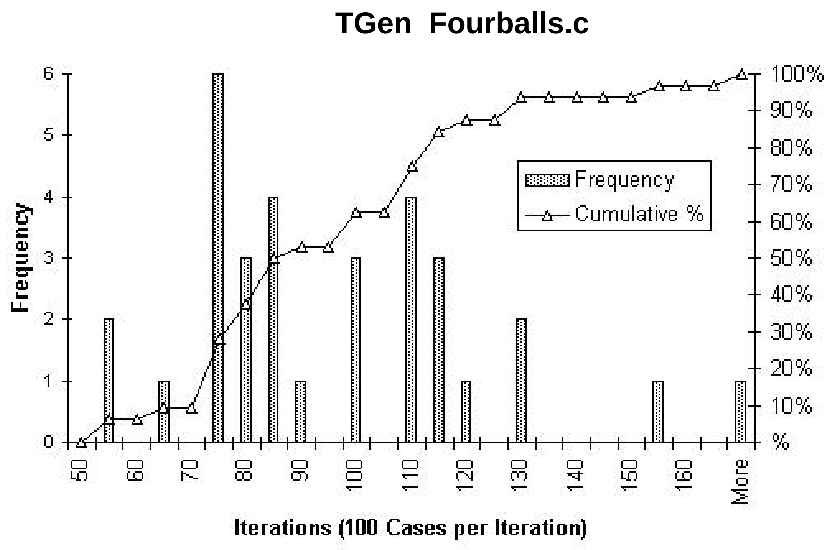


Figure 4: Runs of `TGen` and `Random` for `Bisect.c`.

By comparison, `Random` could not achieve 100% coverage with fewer than 24 iterations. More than half of the `Random` runs required 42 or more iterations to achieve full coverage. The cumulative curve of `Random` is shifted further to the right, indicating that overall, it takes greater effort for `Random` to achieve the goal of 100% coverage.

The differences in the results for `Fourballs.c` are greater. Table 3 shows that `TGen` required an average of 95 iterations to achieve 100% coverage, compared with 159 iterations for `Random`. Median and mode values are similar. In Figure 5, the horizontal axes are not identical: the graph for `TGen` ranges from 50 to 160, whereas the graph for `Random` ranges from 110 to 230. Also, over 90% of the `TGen` runs completed in 130 iterations or less. By contrast the best `Random` run required 110 iterations, with most runs requiring between 150 and 175 iterations. Only eight of the 32 `Random` runs completed in fewer than 150 iterations.

The differences in the results for `Tritype.c` are even greater. Table 3 shows that `TGen` required an average of 145 iterations compared with 1259 iterations for `Random`. Figure 6 shows that only five `Random`



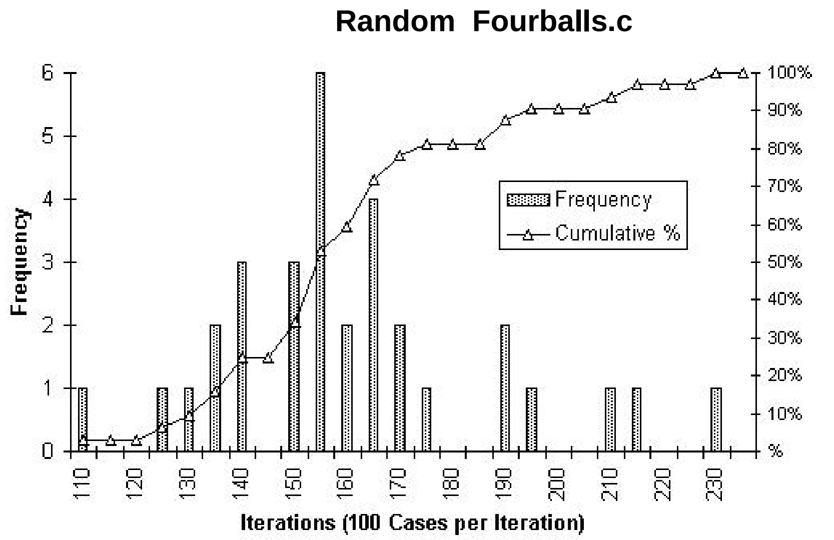
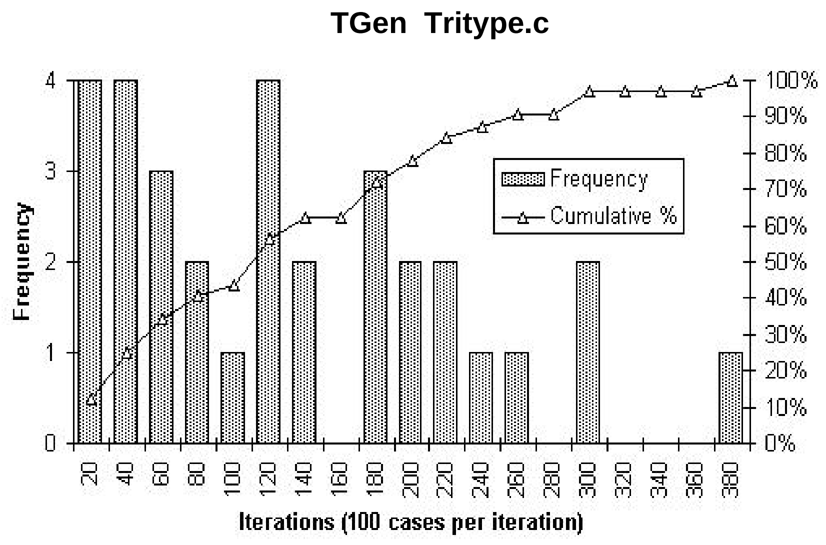


Figure 5: Runs of `TGen` and `Random` for `Fourballs.c`.

runs completed in 150 iterations or less; the rest required 350 iterations or greater, with 12 requiring over 1000 iterations. By contrast, 90% of `TGen` runs completed within 260 iterations. Table 3 also shows results for `TGen` and `Random` runs attempting 100% branch coverage of `Tritype.c`. The results are similar to the statement coverage runs with `TGen` requiring an average of 132 iterations and `Random` requiring an average of 1100 iterations.

The results show that for those source programs with some complexity, `TGen` outperforms `Random` over a number of runs, at times dramatically. For those source programs with little complexity, `TGen` and `Random` perform identically, achieving 100% coverage immediately. `TGen` performs better than `Random` when the source code contains nested conditionals or nested loops in which the predicates are difficult to satisfy.



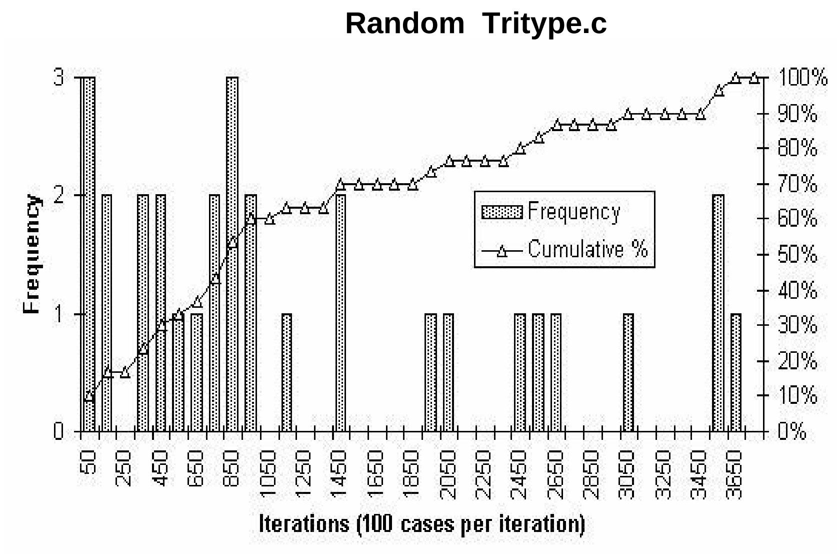
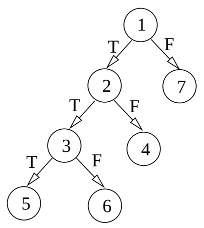


Figure 6: Runs of `TGen` and `Random` for `Tritype.c`.

## 5 Related Work

Other researchers have investigated the use of optimization techniques for test-data generation. Jones et al. [9] developed a genetic algorithm for test-data generation for branch coverage. They use a control-flow graph that represents one, two, and three iterations of each loop; because their representation unrolls each loop a specified number of times, their control-flow graphs are acyclic. A program is instrumented so that as it executes with a test case, it records the branches it reaches and the fitness of that test case. The fitness function uses the branch value, along with the value of the branch condition, to determine the fitness of the test case. The authors implemented the approach and performed experiments with number of small programs.

The approach presented in this paper also uses branch information to evaluate the fitness function, except it uses the control-dependence graph for fitness evaluation, which can given a more precise fitness evaluation than the Jones et al. approach. To see this, consider the following control-flow graph.



Suppose that there were two test cases, $t1$ and $t2$, such that the path through the control-flow graph for $t1$ is 1, 2, 4, and for $t2$ is 1, 7. Furthermore, suppose that node 5 is the target of the search. Under the approach presented in this paper, $t1$ would be given a higher fitness than $t2$ because it has predicate 1T in common with the predicate path of target node 5. Under the Jones et al. approach, $t1$ and $t2$ would be given the same low fitness because neither test case executes the target or one of its siblings (in the control-flow graph); the fact that $t1$ is "closer" to the target than $t2$ is not incorporated into the fitness calculation.

Michael et al. [17] present an approach that is an implementation of Korel's function minimization approach to test-data generation [16] using a genetic algorithm. Korel applied gradient descent to find test data that caused a "branch function" to take on a negative value; these branch functions are developed so that, when the value is negative, the correct branch is selected on a path to a particular target. Ferguson and Korel later added "chaining" to the algorithm to handle cases for which function minimization cannot be applied (e.g., branches based on booleans or other non-arithmetic conditions) or for which the gradient descent method of minimization fails [7].

One goal of this test-data generation approach, as stated by the authors, is to cover *all* branches in a program, which means that all code locations should ultimately be reached (unless they are infeasible). Thus, they delay attempts to satisfy a certain condition until they have found test cases that reach that condition.

The technique presented in this paper has several advantages over Michael et al.'s technique. First, the use of the control-dependence graph to identify predicate paths for the target and the test case lets the algorithm quickly, and more efficiently than applying function minimization, compute the fitness of a test case. Second, the approach does not require coverage of all test requirements in the program. Rather, the approach can generate test data for single or multiple sets of targets, which makes it more useful for applications, such as regression testing.

Finally, Tracey et al. [21] use another optimization technique, simulated annealing, to generate test data, and performed a few small experiments. They state that to compare their approach to one using genetic algorithms would require implementation of a genetic-algorithm approach. However, neither of their approaches has been implemented. Thus, there is no data that could be used to compare the efficiency and effectiveness of simulated annealing approaches and genetic algorithm approaches. Future work could develop implementations of, and experiment with, simulated annealing and genetic algorithm approaches to test-data generation.

## 6 Conclusions

This paper presents GenerateData, an algorithm for automatic generation of test data for a given program. GenerateData uses a search heuristic called a genetic algorithm directed by the control-dependence graph of the program. The simplicity of the genetic algorithm and the direction provided by the use of the control-dependence graph combine to produce an effective tool for test-data generation. Parallel processing is also employed in the implementation of GenerateData for additional computational power to speed up what could possibly be a lengthy search.

Currently, the prototype, TGen, is implemented for statement and branch coverage. Results of experiments with TGen (the implementation of the algorithm GenerateData used in this study) show that for programs with little complexity, TGen and Random provide statement and branch coverage immediately. With programs of some complexity, TGen outperforms Random, at times significantly. At the heart of the success of TGen is the use of the control-dependence graph to direct the search for test cases. At different points in time during execution, the implementation targets different test requirements (currently statements and branches, eventually paths and definition-use pairs) and uses the control-dependence graph to help form new test cases. This procedure may be superior to random methods of generating test cases, and possibly to other goal-directed methods that require more time-consuming analyses of the source code to generate new test cases.

The use of parallel processing improved overall execution time almost linearly. This is because TGen is designed to distribute the most time-consuming task, that of executing the instrumented program on a single test case, among as many processors as are available. The manner of distribution is round robin, which automatically balances the total load.

TGen is useful for regression testing, particularly when the original test suite is available. TGen allows the user to target for coverage only the new or modified sections of code, thus instructing TGen to ignore those sections already tested. This targeting of code concentrates TGen's full attention and computing power only to that section of code of interest. Furthermore, the original test suite may be used as part of the initial population, which is beneficial because the original test suite should contains a variety of test cases that cover the requirements of the original program. Thus, the test suite has good genetic material with which TGen can work.

This paper has outlined ways to extend TGen to provide path and definition-use coverage. Once path and definition-use coverage are restated as problems of covering *sets* of statements, the TGen approach can be applied. The paper also outlined a straightforward way to use TGen for programs with multiple procedures. This is possible because of the ease with which TGen can focus on specific sections of code. TGen's evaluation function can be modified to give much greater weight to those test cases which cover the calls to the procedures of interest. Thus, TGen can ignore the majority of the code and work only with test cases that execute the procedure being analyzed.

GenerateData represents a potentially powerful approach in the area of automatic test-data generation. Its power lies in the fact that the generation of new test cases is both simple and yet goal-directed. The results presented here, however, are preliminary. GenerateData has been tested on only six programs and, although the results are promising, more experimentation must be done before any conclusive statements can be made. As more is learned about the strengths and weaknesses of TGen, a metric that can be applied to a program and that will predict the likely success of an application of TGen on the program can be developed. It seems probable, however, that `GenerateData`'s real potential is not in generating test data for small programs, such as those described in this paper. Rather, `TGen`'s real strength is in working with large programs, particularly on programs with multiple procedures. It is on these programs that the guided search capability of `TGen`'s approach will be fully utilized.

Another possibility is to couple `TGen` with other, possibly exhaustive, search techniques to see if cooperation among different search algorithms is possible. The implementation of `TGen` can be extended to handle test-data generation for multiple procedures. Finally, the implementation will be extended to allow path and definition-use coverage.

## References

[1] R. S. Boyer, B. Elspas, and K. N. Levitt. Select - A formal system for testing and debugging programs by symbolic execution. Proceedings of the International Conference on Reliable Software, June 1975.

[2] K-H. Chang, J. H. Cross, W. H. Carlisle, and D. B. Brown. A framework for intelligent test data generators. Journal of Intelligent and Robotic Systems - Theory and Applications, July 1991.

[3] L. A. Clarke. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, 2(3):215–222, September 1976.

[4] L. Davis. Genetic Algorithms and Simulated Annealing. Morgan Kaufmann, 1987.

[5] L. Davis. Handbook of Genetic Algorithms. Van Nostrand Reinhold, New York, 1991.

[6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering, 17(9):900–910, September 1991.

[7] R. Ferguson and B. Korel. The chaining approach for software test data generation. ACM TOSEM, vol. 5, no. 1, pages 63–86, January 1996.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. TOPLAS, vol. 9, no. 3, pages, 319–349, July 1987.

[9] B. Jones, H. Sthamer, D. Eyres. Automatic Structural Testing Using Genetic Algorithms. Software Engineering Journal 11(5), September 1996, pp. 299–306.

[10] A. Geist, A. Beguelin, J. Dongarra, W. C. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994, Cambridge, MA.

[11] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of Software Engineering. Prentice Hall, Englewood Cliffs, NJ, 1991.

[12] D. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, Massachusetts, 1989.

[13] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools, OSU-CISRC-3/97-TR17,The Ohio State University, March 1997.

[14] J. Holland. Adaptation in natural and artificial systems. Ann Arbor: The University of Michigan Press, 1975.

[15] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering 3(4): 266–278, July 1977.

[16] B. Korel. Automated software test data generation. IEEE Transactions on Software Engineering, 16(8):870–879, August 1990.

[17] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton.. Genetic Algorithms for Dynamic Test Data Generations. Technical Report RSTR-003-97-11, May 1997.

[18] H. D. Mills, M. D. Dyer, and R. C. Linger. Cleanroom software engineering. IEEE Software 4(5): 19–25, September 1987.

[19] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. IEEE Transactions on Software Engineering, 2(4):293–300, December 1976.

[20] P. Thévenod-Fosse, H. Waeselynck. STATEMATE: Applicd to Statistical Software Testing. ACM SIGSOFT Proceedings of the 1993 International Symposium on Software Testing and Analysis, Software Enginering Notes 23(2), pp. 78–81, June 1993.

[21] N. Tracey, J. Clark, K. Mander. Automated Program Flaw Finding Using Simulated Annealing. ACM SIGSOFT Proceedings of the 1998 International Symposium on Software Testing and Analysis, March 1998.

[22] J. M. Voas, L. Morell, and K. W. Miller. Predicting where faults can hide from testing. IEEE Software, 8(2), 41–48, March 1991.