Generating Software Test Data by Evolution

Christoph C. Michael, *Member*, *IEEE*, Gary McGraw, *Member*, *IEEE*, and Michael A. Schatz

Abstract—This paper discusses the use of genetic algorithms (GAs) for automatic software test data generation. This research extends previous work on dynamic test data generation where the problem of test data generation is reduced to one of minimizing a function [1], [2]. In our work, the function is minimized by using one of two genetic algorithms in place of the local minimization techniques used in earlier research. We describe the implementation of our GA-based system and examine the effectiveness of this approach on a number of programs, one of which is significantly larger than those for which results have previously been reported in the literature. We also examine the effect of program complexity on the test data generation problem by executing our system on a number of synthetic programs that have varying complexities.

Index Terms—Software testing, automatic test case generation, code coverage, genetic algorithms, combinatorial optimization.

1 Introduction

An important aspect of software testing involves judging how well a series of test inputs tests a piece of code. Usually, the goal is to uncover as many faults as possible with a potent set of tests since a test series that has the potential to uncover many faults is obviously better than one that can only uncover a few. Unfortunately, it is almost impossible to predict how many faults will be uncovered by a given test set. This is not only because of the diversity of the faults themselves, but because the very concept of a *fault* is only vaguely defined (c.f., [3]). Still, it is useful to have some standard of test adequacy, to help in deciding when a program has been tested thoroughly enough. This leads to the establishment of *test adequacy criteria*.

Once a test adequacy criterion has been selected, the question that arises next is how to go about creating a test set that is *good* with respect to that criterion. Since this can be difficult to do by hand, there is a need for *automatic test data generation*.

Unfortunately, test data generation leads to an undecidable problem for many types of adequacy criteria. Insofar as the adequacy criteria require the program to perform a specific action, such as reaching a certain statement, the halting problem can be reduced to a problem of test data generation. To circumvent this dilemma, test data generation algorithms use heuristics, meaning that they do not always succeed in finding an adequate test input. Comparisons of different test data generation schemes are usually aimed at determining which method can provide the most benefit with limited resources.

In this paper, we introduce GADGET (the Genetic Algorithm Data GEneration Tool), which uses a test data generation paradigm commonly known as *dynamic test data generation*. Dynamic test data generation was originally proposed by [1] and then investigated further by [2], [4], and [5]. During dynamic test generation, the source code of a program is instrumented to collect information about the program as it executes. The resulting information, collected during each test execution of the program, is used to heuristically determine how close the test came to satisfying a specified test requirement. This allows the test generator to modify the program's input parameters gradually, nudging them ever closer to values that actually do satisfy the requirement. In essence, the problem of generating test data reduces to the well-understood problem of function minimization.

The approach usually proposed for performing this minimization is gradient descent, but gradient descent suffers from some well-known weaknesses. Thus, it is appealing to use more sophisticated techniques for function minimization, such as genetic search [6], simulated annealing [7], or tabu search [8]. In this paper, we investigate the use of genetic search to generate test cases by function minimization.

In the past, automatic test data generation schemes have usually been applied to simple programs (e.g., mathematical functions) using simple test adequacy criteria (e.g., branch coverage). Random test generation performs adequately on these problems. Nevertheless, it seems unlikely that a random approach could also perform well on realistic test-generation problems, which often require an intensive manual effort. Indeed, our results suggest that random test generation performs poorly on realistic programs. The broader implication is that, due to their simplicity, toy programs fail to expose the limitations of some test-data generation techniques. Therefore, such programs provide limited utility when comparing different test generation methods. Because GADGET was designed to work on large programs written in C and C++, it is possible for us to examine the effects of program complexity on the difficulty of test data generation.

We examine a feature of the dynamic test generation problem that does not have an analog in most other function minimization problems. If we are trying to satisfy many test requirements for the same software, we have to perform many function minimizations, but the functions being minimized are sometimes quite similar. That makes it

• *The authors are with Cigital Corporation, Suite 400, 21351 Ridgetop Circle, Dulles, VA 20166. E-mail: {ccmich, gem, mascha}@cigital.com.*

Manuscript received 17 Dec. 1997; revised 5 Feb. 1999; accepted 24 Oct. 2000. Recommended for acceptance by D. Rosenblum.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106067.

possible to solve one problem by coincidence while trying to solve another. In other words, the test generator can find inputs that satisfy one requirement even though it is searching for inputs to satisfy a different one.

On the larger programs we tested, coincidental discovery of test inputs satisfying new requirements was much more common than their deliberate detection (the GAs often satisfied one test requirement while they were trying to satisfy a different one). In fact, the ability of test data generators to satisfy coverage requirements coincidentally seems to play an important role in determining their effectiveness.

Random test generation did not perform well in our experiments. Moreover, our empirical results show an increasing performance gap between random test generation and the more sophisticated test generation methods when tests are generated for increasingly complex programs. This suggests that the additional effort of implementing more sophisticated test generation techniques is ultimately justified.

This paper begins with an overview of automatic test data generation methods (Section 2), followed by an introduction to genetic algorithms in the context of test-data generation (Section 3). In Section 4, we describe our own test-data generation system, GADGET. Finally, in Section 5, we empirically examine the performance of our system on a number of test programs.

2 Test Adequacy Criteria and Test Data Generation

Some test paradigms call for inputs to be selected on the basis of test adequacy criteria, which are used to ensure that certain features of the source code are exercised (in testing terminology, these features are to be *covered* by the test inputs). Some studies, such as [9], [10], [11], have concluded that test adequacy does, in fact, improve the ability of a test suite to reveal faults, though [12], [13], [14], [15], among others, describe situations where this is not true. Whether or not test adequacy criteria really measure the quality of a test suite, they are an objective way to measure the thoroughness of testing.

These benefits cannot be realized unless adequate test data (i.e., test data that satisfy the adequacy criteria) can be found. Manual generation of such tests can be quite time-consuming, so it would be appealing to have algorithms that can examine a program's structure and generate adequate tests automatically.

It is desirable to have test data generation algorithms that are more powerful in the sense of being more capable of finding adequate tests. Our research addresses this need.

2.1 Code Coverage and Test Adequacy Criteria

Many test adequacy criteria require certain features of a program's source code to be exercised. A simple example is a criterion that says, "Each statement in the program should be executed at least once when the program is tested." Test methodologies that use such requirements are usually called *coverage analyses* because certain features of the source code are to be covered by the tests. A test adequacy criterion generally leads to a set of *test requirements* specifically stating the conditions that the tests must fulfill. For example, each statement in a program might be associated with a requirement asking that the statement in question be executed during testing.

The example given above describes *statement coverage*. A slightly more refined approach is *branch coverage*. This criterion requires every conditional branch in the program to be taken at least once. For example, supposing we want to obtain branch coverage of the following code fragment:

```
if (a >= b) { do one thing }
else        { do something else }
```

we must satisfy two test requirements: There must be one program input that causes the value of the variable `a` to be greater than or equal to the value of `b`, and there must be one that causes the value of `a` to be less than that of `b`. One effect of these requirements is to ensure that both the "do one thing" and "do something else" sections of the program are executed.

There is a hierarchy of increasingly complex coverage criteria having to do with the conditional statements in a program. We shall refer to this hierarchy as defining *levels* of coverage. At the top of the hierarchy is *multiple condition coverage*, which requires the tester to ensure that every permutation of values for the Boolean variables in every condition occurs at least once. At the bottom of the hierarchy is function coverage, which requires only that every function be called once during testing (saying nothing about the code inside each function). Somewhere between these extremes is *condition-decision coverage*, which is the criterion we use in our test-data generation experiments.

A *condition* is an expression that evaluates to TRUE or FALSE, but does not contain any other TRUE/FALSE-valued expressions, while a *decision* is an expression that influences the program's flow of control. To obtain condition-decision coverage, a test set must make each condition evaluate to TRUE for at least one of the tests and each condition evaluate to FALSE for at least one of the tests. Furthermore, the TRUE and FALSE branches of each decision must be exercised. Put another way, condition-decision coverage requires that each branch in the code be taken *and* that every condition in the code be TRUE at least once, and FALSE at least once.

With any of these coverage criteria, we must ask what to do when an existing test set fails to meet the chosen criterion. In many cases, the next step is to try to find a test set that *does* satisfy the criterion. Since it can be quite difficult to manually search for test inputs satisfying certain requirements, test data generation algorithms are used to automate this process.

2.2 Previous Work in Test Data Generation

The term "test generation" is commonly applied to a number of diverse techniques. For example, tests may be generated from a specification (in order to exercise features of the specification), or they may be generated from state model of software operation (in order to exercise various states or combinations of states).

For a program with a complex graphical user interface, test generation may simply consist of finding tests that

exercise all aspects of the interface. On the other hand, many software systems package a diverse collection of services and, for such packages, it is often considered sufficient to ensure that most or all services are used during testing. In such cases, it is often straightforward to find inputs that exercise a given feature and, so, a test generator only has to list the features or combinations of features that are to be tested. In other words, the term "test data generation" sometimes only refers to the process of coming up with concrete test criteria.

Unfortunately, some test criteria are harder to satisfy. If we want to satisfy *code coverage* criteria or exercise some other precisely defined aspect of program semantics, it may be far from obvious what program inputs satisfy a given criterion. This paper is concerned with this case: We are *given* a set of test adequacy criteria and the goal is to find test inputs that cause the criteria to be satisfied.

There are many existing paradigms for this type of automatic test data generation. Perhaps the most commonly encountered are random test data generation, symbolic (or path-oriented) test data generation, and dynamic test data generation. In the next three sections, we will describe each of these techniques in turn. The GADGET system we describe in this paper is a dynamic test generator. In our experiments, we use random data generation as a baseline for comparison.

2.2.1 Random Test Data Generation

Random test data generation simply consists of generating inputs at random until a useful input is found. The problem with this approach is clear: With complex programs or complex adequacy criteria, an adequate test input may have to satisfy very specific requirements. In such a case, the number of adequate inputs may be very small compared to the total number of inputs, so the probability of selecting an adequate input by chance can be low.

This intuition is confirmed by empirical results (including those reported in Section 5). For example, [16] found that random test generation was outperformed by other methods, even on small programs where the goal was to obtain statement coverage. More complex programs or more complex coverages are likely to present even greater problems for random test data generators. Nonetheless, random test data generation makes a good baseline for comparison because it is easy to implement and commonly reported in the literature.

2.2.2 Symbolic Test Data Generation

Many test data generation methods use symbolic execution to find inputs that satisfy a test requirement (e.g., [17], [18], [19]). Symbolic execution of a program consists of assigning symbolic values to variables in order to come up with an abstract, mathematical characterization of what the program does. Thus, ideally, test data generation can be reduced to a problem of solving an algebraic expression.

A number of problems are encountered in practice when symbolic execution is used. One such problem arises in indefinite loops, where the number of iterations depends on a nonconstant expression. To obtain a complete picture of what the program does, it may be necessary to characterize what happens if the loop is never entered, if it iterates once, if it iterates twice, and so on *ad infinitum*. In other words, the symbolic execution of the program may require an infinite amount of time.

Test data generation algorithms solve this problem in a straightforward way: The program is only executed symbolically for one control path at a time. Paths may be selected by the user, by an algorithm, or they may be generated by a search procedure. If one path fails to result in an expression that yields an adequate test input, another path is tried.

Loops are not the only programming constructs that cannot easily be evaluated symbolically; there are other obstacles to a practical test data generation algorithm based on symbolic execution. Problems can arise when data is referenced indirectly, as in the statement:

```
a = B[c+d]/10
```

Here, it is unknown which element of the array `B` is being referred to by `B[c+d]` because the variables `c` and `d` are not bound to specific values.

Pointer references also present a problem because of the potential for aliasing. Consider that the C code fragment:

```
*a = 12;
*b = 13;
c = *a;
```

results in `c` taking the value `12` unless the pointers `a` and `b` refer to the same location, in which case, `c` is assigned the value `13`. Since `a` and `b` are not bound to numeric values during symbolic execution, the final value in `c` cannot be determined.

Technically, any computable function can be computed without the use of pointers or arrays, but it is not normal practice to avoid these constructs when writing a program. Thus, although array and pointer references are not a theoretical impediment to the use of symbolic execution, they complicate the problem of symbolically executing real programs.

2.2.3 Dynamic Test Data Generation

A third class of test data generation paradigms is dynamic test data generation, introduced in [1] and exemplified by the TESTGEN system of [2], [16], as well as the ADTEST system of [5]. This paradigm is based on the idea that if some desired test requirement is not satisfied, data collected during execution can be still used to determine which tests come *closest* to satisfying the requirement. With the help of this feedback, test inputs are incrementally modified until one of them satisfies the requirement.

For example, suppose that a hypothetical program contains the condition

```
if (pos >= 21) ...
```

on line 324 and that the goal is to ensure that the TRUE branch of this condition is taken. We must find an input that will cause the variable `pos` to have a value greater than or equal to 21 when line 324 is reached. A simple way to determine the value of pos on line 324 is to execute the program up to line 324 and then record the value of `pos`.

Let $pos_{324}(x)$ denote the value of `pos` recorded on line 324 when the program is executed in the input x. Then, the function

$$\Im(x) = \begin{cases} 21 - pos_{324}(x), & \text{if } pos_{324}(x) < 21; \\ 0, & \text{otherwise} \end{cases}$$

is minimal when the TRUE branch is taken on line 324. Thus, the problem of test data generation is reduced to one of function minimization: To find the desired input, we must find a value of x that minimizes $\Im(x)$.

In a sense, the function $\Im$ (which we will also call an *objective function*) tells the test generator how close it is to reaching its goal. If x is a program input, then the test generator can evaluate $\Im(x)$ to determine how close x is to satisfying the test requirement currently being targeted. The idea is that the test generator can now modify x and evaluate the objective function again in order to determine what modifications bring the input closer to meeting the requirement. The test generator makes a series of successive modifications to the test input using the objective function for guidance and, hopefully, this leads to test that satisfies the requirement (in fact, $\Im$ can only be said to provide heuristic information, as will become apparent when we discuss the construction of $\Im$ in Section 4.3).

In the TESTGEN system of [2], the minimization of $\Im(x)$ begins by establishing an overall goal, which is simply the satisfaction of a certain test requirement. The program is executed on a seed input, and its behavior on this input is used as the basis of a search for a satisfactory input (that is, if the seed input is not satisfactory itself).

The subsequent action depends on whether the the execution reaches the section(s) of code where the test requirement is supposed to hold (for example, whether it reaches line 324 in the example above). If it does, then function minimization methods can be used to find a useful input value.

If the code is not reached, a subgoal is created to bring about the conditions necessary for function minimization to work. The subgoal consists of redirecting the flow of control so that the desired section of code *will* be reached. The algorithm finds a branch that is responsible (wholly or in part) for directing the flow of control away from the desired location and attempts to modify the seed input in a way that will force the control of execution in the desired direction.

The new subgoal can be treated in the same way as other test requirements. Thus, the search for an input satisfying a subgoal proceeds in the same way as the search for an input satisfying the overall goal. Likewise, more subgoals may be created to satisfy the first subgoal. This recursive creation of subgoals is called *chaining* in [2] and [4].

Korel's approach is advantageous when there is more than one path that reaches the desired location in the code. The test data generation algorithm is free to choose whichever path it wants (as long as it can force that path to be executed), and some paths may be better than others. For example, suppose we want to take the TRUE branch of the condition

```
if (b > 10) ...
```

but suppose that `b` has a default value of 3. It may be that one execution path gives `b` a new value, while a different path simply leaves the value of `b` alone. As long as we only take the path that leaves `b` with its default value, we will never be able to make the condition TRUE; no choice of inputs can make the default value of `b` be anything other than 3. Therefore, the test generation algorithm must know how to select the execution path that assigns a new value to `b`. In the TESTGEN system, heuristics are used to select the path that seems most likely to have an impact on the target condition.

In the ADTEST system of [5], an entire path is specified in advance, and the goal of test data generation is to find an input that executes the desired path. Since it is known which branch must be taken for each condition on the path, all of these conditions can be combined in a single function whose minimization leads to an adequate test input. For example, if the desired path requires taking the TRUE branch of the condition

```
if (b >= 10) ...
```

on line 11 and taking the FALSE branch of the condition

```
if (c >= 8)
```

on line 13, then one can find an adequate test input by minimizing the function $\Im_1(x) + \Im_2(x)$, where

$$\Im_1(x) = \begin{cases} 10 - b_{11}, & \text{if } b_{11} < 10 \text{ on line 11}; \\ 0, & \text{otherwise} \end{cases}$$

$$\Im_2(x) = \begin{cases} c_{13} - 8, & \text{if } c_{13} > 8 \text{ on line 13}; \\ 0, & \text{otherwise}. \end{cases}$$

(Here, c_{13} and b_{11} are actually functions of the input value x.)

Unfortunately, this function cannot be evaluated until line 10 and line 13 are both reached. Therefore, the ADTEST system begins by trying to satisfy the first condition on the path, adding the second condition only after the first condition has been satisfied. As more conditions are reached, they are incorporated in the function that the algorithm seeks to minimize.

Another test generation system relevant to our work is the QUEST/Ada system of [20], [21]. This is a hybrid system combining random testing and dynamic testing for Ada code. Once the code is instrumented and ranges and types of input variables have been provided, the system creates test data using rule-based heuristics. For example, values of parameters are adjusted according to one such rule to increase or decrease by a fixed constant percentage. The test adequacy criterion chosen by Chang et al. is branch coverage. The system creates a coverage table for all branches and marks those that have been successfully covered. Table 1 provides an example of such a branch table. The table is consulted during analysis to determine which branches to target for testing. Partially covered branches are always chosen over completely noncovered branches.

Although QUEST/Ada does not use the dynamic test generation paradigm we have been describing, the coverage table of [21] is relevant to our research because it provides a strategy for dealing with the situation where a desired condition is not reached. Instead of picking a particular

TABLE 1
A Sample Coverage Table after Chang

	Branch	
Decision	TRUE	FALSE
1	X	X
2	X	X
3	X	-
4	X	-
5	-	X
6	-	-

This table is generated from a program flow chart. The table provides information regarding the covered branches and directs future test case generation. We adapt this approach for our generator as well.

condition, as TESTGEN does, or picking a particular path, like ADTEST, this strategy is opportunistic and seeks to cover whatever conditions it can reach. Although this is inefficient when one only wants to exercise a certain feature of the code under test, it can save quite a bit of unnecessary work if one wants to obtain complete coverage according to some criterion.

We developed the coverage-table strategy independently and use it in our test-generation system.

In [22], [23], [24], simulated annealing is used in conjunction with dynamic test generation in much the same way that we use genetic algorithms. These papers only report results for small programs, but they show how dynamic test generation can be applied numerous test generation problems other than the satisfaction of structural coverage criteria.

The GADGET test generation system, which we discuss in this paper, is a dynamic test generation system like TESTGEN and ADTEST, but it uses genetic search to perform optimization, instead of the gradient descent techniques used by TESTGEN and ADTEST; the advantages of this will be discussed in Section 3.1. In [25], we presented some preliminary results on the performance of an early prototype and, in [26], we examined the performance of the GADGET system using a number of different optimization techniques, including genetic search and simulated annealing.

2.3 Contributions of This Paper

The research described in this paper addresses two limitations commonly found in dynamic test-data generation systems. First, many systems make it difficult to generate tests for large programs because they only work on simplified programming languages. Second, many systems use gradient descent techniques to perform function minimization and, therefore, they can stall when they encounter local minima (this problem is described below in greater detail).

Limited program complexity is a drawback of the TESTGEN. It can only be used with programs written in a subset of the Pascal language. Aside from problems of practicality, the problem with such limitations is that they prevent one from studying how the complexity of a program affects the difficulty of generating test data. The unchallenging demands of simple programs can make simple schemes like random test generation appear to work better in comparison to other methods than they actually do.

Both TESTGEN and the QUEST/Ada system use gradient descent to minimize the objective function. This technique is limited by its inability to recognize local minima in the objective function (see Section 3.1). Our main goal is to overcome this limitation.

In this paper, we report on GADGET, a test generation system designed for programs written in C or C++. GADGET automatically generates test data for arbitrary C/C++ programs, with no limitations on the permissible language constructs and no requirement for hand-instrumentation. (However, GADGET does not generate meaningful character strings unless those strings represent numbers, and it does not generate meaningful values for compound data-types, even though such values can be regarded as inputs if they are read from an external file.)

We report test results for a program containing over 2,000 lines of source code, excluding comments. To our knowledge, this is the largest program for which results have been reported. (Although [5] reported that their system had been run on programs as large as 60,000 lines of source code, no results were presented.) The ability to generate tests for programs using all C/C++ constructs has the added benefit of allowing us to study the effects of program complexity on the difficulty of test data generation. Some experimental results were also presented in [26], but the current paper also presents a self-contained explanation of GADGET's underlying test-generation paradigm.

The GADGET system uses genetic algorithms to perform the function minimization needed during dynamic test data generation. In this respect, it differs from the TESTGEN and ADTEST systems, which use gradient descent. The advantage of using genetic algorithms is that they are less susceptible to local minima, which can cause a test-generation algorithm to halt without finding an adequate input.

Genetic algorithms were used in a different way by [27]. That system judges test inputs according to how "interesting" they are, according to user-defined criteria for what is interesting. Thus, although that system is used for generating software tests, it is a test generator in a different sense than our system. It does not strive to satisfy specific test requirements and, thus, it does not need to use semantic information about the target program (in contrast, semantic information is crucial for the other test generation techniques we have discussed).

The most frequently cited advantage of genetic algorithms, when they are compared to gradient descent methods, is that genetic algorithms are less likely to stall in a local minimum—a portion of the input space where $\Im(x)$ appears to be minimal but is not. There is also a second advantage when several paths to the desired location are available. Unlike gradient descent methods, which must concentrate on a single path, the implicit parallelism of genetic algorithms allows them to examine many paths at once. This presents a partial solution to the path-selection problem described in Section 2.2.3.

Certain limitations are common to all dynamic test generation systems, including our own. Existing systems are limited to programs whose inputs are scalar types

(though the nature of dynamic test generation systems allows the use of arbitrary data-types within the program itself, which is a further advantage of dynamic test-data generation over the symbolic methods described in Section 2.2.2).

The main challenge posed by nonscalar program inputs is that, often, the data members must satisfy certain constraints for the input to make sense. For example, the characters in an array may be required to spell out the name of a file, an integer may represent the number of elements in a list stored elsewhere in the same object instantiation, etc. What these constraints actually are can vary a great deal from one application to the next and, so, it is difficult to automate universal support for them. (Of course, violating such constraints can make for interesting test cases, but, if many input values are nonsensical most or all of the time, it can lead to poor coverage.) In such cases, users must build auxiliary functions that accept scalar inputs from the test generator and that construct meaningful values for variables having complex data types.

Existing dynamic test generation techniques are also somewhat unintelligent in their handling of TRUE/FALSE-valued variables or enumerated types. Programs using such variables within conditional statements do not seem to have been used in past research. (The problem presented by such variables will be examined in Section 5.4.)

Dynamic test generation has the advantage of treating a program somewhat as a black box, which makes most programming constructs transparent to the algorithm. It is only necessary that we be able to extract the information needed for calculating the objective function and for determining whether the path of execution leads to the location where the objective function is evaluated. This gives dynamic test generation the flavor of a general-purpose test generation technique that can be applied automatically to a wide range of programs. The GADGET system advances this conception of test generation in two ways. First, it uses genetic search, which is a general-purpose optimization strategy and, unlike gradient descent, does not assume an absence of local minima in the objective function. Second, GADGET uses automatic code instrumentation, which eliminates a practical obstacle when generating tests for large programs. This allows us to easily examine the effects of program complexity on the performance of test data generators, as we do in this paper.

3 Genetic Algorithms for Test Data Generation

In dynamic test data generation, the problem of finding software tests is reduced to an optimization problem. Most existing techniques solve the optimization problem with gradient descent,[1] but this is not a general-purpose approach because of certain assumptions it makes about the shape of the objective function, which we will discuss momentarily. This is the motivation for building the system we describe in this paper, which replaces gradient descent with a more general optimization method, namely, genetic search.

1. The exceptions are GADGET itself, for which preliminary results were published in [25], and the system described in [22], which was published after the original preparation of this paper.

3.1 Function Minimization

Recall that, in dynamic test generation, the target program is repeatedly executed on a series of test inputs. This is done to evaluate an objective function $\mathfrak{F}$, which heuristically tells the test generator how close each input has come to satisfying the test requirement that is currently targeted. This lets the test generator make successive modifications to the test input, bringing it ever closer to satisfying the requirement. This process is equivalent to conducting a minimization of the function $\mathfrak{F}$.

Thus, the ability to numerically minimize a function value is the key to dynamic test data generation. The standard minimization technique, used by [1], [2], [5], is gradient descent. It essentially works by making successive changes to a test input in such a way that each change decreases the value of the objective function. Insofar as the objective function tells us how close the input is to satisfying the selected test requirement, each modification brings the input closer to meeting that requirement.

The minimization method suggested in [16] is a form of gradient descent. Small changes in one input value are made initially to determine a good direction for making larger moves. When an appropriate direction is found, increasingly large steps are taken in that direction until no further improvement is obtained (in which case, the search begins anew with small input modifications), or until the input no longer reaches the desired location (in which case, the move is tried again with a smaller step size). When no further progress can be made, a different input value is modified, and the process terminates when no more progress can be made for any input value.

Reference [5] also uses gradient descent; specifically, the system described there uses a quasi-Newton technique (see [28]).

Gradient descent can fail if a local minimum is encountered. A local minimum occurs when none of the changes of input values that are being considered lead to a decrease in the function value and, yet, the value is not globally minimized. The problem arises because it is only possible to consider a limited number of input values (i.e., a small section of the search space) due to resource limitations. The input values that are considered may suggest that any change of values will cause the function's value to increase, even when the current value is not truly minimal. This situation is illustrated in Fig. 1.

In other areas where optimization is used, the problem of local minima has led to the development of function minimization methods that do not blindly pursue the steepest gradient. Notable among these are simulated annealing [7], tabu search [8], [29], and genetic algorithms [6], [30], [31]). In this paper, we apply two genetic algorithms to the problem of test data generation.

3.2 Genetic Algorithms

A genetic algorithm (GA) is a randomized parallel search method based on evolution. GAs have been applied to a variety of problems and are an important tool in machine learning and function optimization. References [30] and [31] give thorough introductions to GAs and provide lists of possible application areas. The motivation behind genetic algorithms is to model the robustness and flexibility of natural selection.

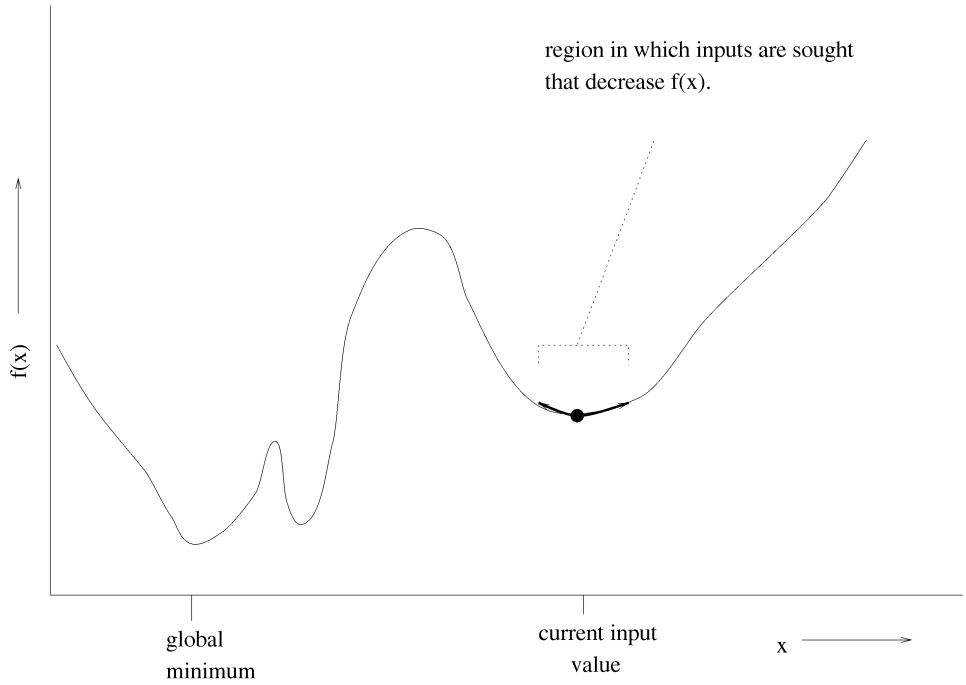


Fig. 1. Illustration of a local minimum. An algorithm tries to find a value of x that will decrease $f(x)$, but there are no such values in the immediate neighborhood of the current x. The algorithm may falsely conclude that it has found a global minimum of f.

In a classical GA, each of a problem's parameters is represented as a binary string. Borrowing from biology, an encoded parameter can be thought of as a gene, where the parameter's values are the gene's alleles. The string produced by the concatenation of all the encoded parameters forms a genotype. Each genotype specifies an individual which is in turn a member of a population. The GA starts by creating an initial population of individuals, each represented by a randomly generated genotype. The fitness of individuals is evaluated in some problem-dependent way, and the GA tries to evolve highly fit individuals from the initial population. In our case, individuals are more fit if they seem closer to satisfying a test requirement; for example, if the goal is to make the value of the variable `pos` greater than or equal to 21 on line 324, then an input that results in `pos` having the value 20 on line 324 is considered more fit than an input that gives it the value -67.

The genetic search process is iterative: evaluating, selecting, and recombining strings in the population during each iteration (generation) until reaching some termination condition occurs. (In our case, a success leads to termination of the search, as does a protracted failure to make any forward progress. This is a relatively common arrangement.)

The basic algorithm, where $P(t)$ is the population of strings at generation t, is:

initialize $P(t)$
evaluate $P(t)$
while = (termination condition not satisfied) **do**
    select $P(t+1)$ **from** $P(t)$
    recombine $P(t+1)$
    evaluate $P(t+1)$
    $t = t+1$

In the first step, evaluation, the fitness of each individual is determined. Evaluation of each string (individual) is based on a fitness function that is problem-dependent. Determining fitness corresponds to the environmental determination of survivability in natural selection, and, in our case, it is determined by the fitness function described in Section 2.2.3.

The next step, selection, is used to find two individuals that will be mated to contribute to the next generation. Selection of a string depends on its fitness relative to that of other strings in the population. Most often, the two individuals are selected at random, but each individual's probability of being chosen is proportional to its fitness. This is known as roulette-wheel selection. Thus, selection is done on the basis of relative fitness. It probabilistically culls from the population individuals having relatively low fitness.

The third step is crossover (or recombination), which fills the role played by sexual reproduction in nature. One type of simple crossover is implemented by choosing a random point in a selected pair of strings (encoding a pair of solutions) and exchanging the substrings defined by that point, as shown in Fig. 2.

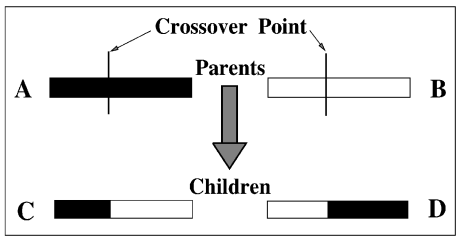


Fig. 2. Single-point crossover of the two parents A and B produces the two children C and D. Each child consists of parts from both parents leading to information exchange.

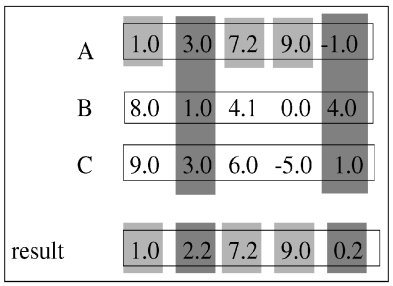


Fig. 3. An illustration of how individuals are combined by a differential GA. Three individuals are chosen as parents and each element of the resulting individual is either copied from the first parent or combined from elements of all three parents.

In addition to evaluation, selection, and recombination, genetic algorithms use mutation to guard against the permanent loss of gene combinations. Mutation simply results in the flipping of bits within a genome, but this flipping of bits only occurs infrequently.

The individuals in the population act as a primitive memory for the GA. Genetic operators manipulate the population, usually leading the GA away from unpromising areas of the search space and toward promising ones, without the GA having to *explicitly* remember its trail through the search space [32].

It is easiest to understand GAs in terms of function optimization. In such cases, the mapping from genotype (string) to phenotype (point in search space) is usually trivial. For example, in order to optimize the function $f(x) = x$, individuals can be represented as binary numbers encoded in normal fashion. In this case, fitness values would be assigned by decoding the binary numbers. As crossover and mutation manipulate the strings in the population, thereby exploring the space, selection probabilistically filters out strings with low fitness, exploiting the areas defined by strings with high fitness. Since the search for individuals with higher fitness is not restricted to a localized region of the objective function, this search technique is not subject to the problems associated with local minima, which were described above.

3.2.1 Differential Genetic Search

A second genetic algorithm that we have also used for generating software tests is the *differential GA* described in [33]. Here, an initial population is constructed as above. Recombination is accomplished by iterating over the inputs in the population. For each such input I, three mates, A, B, and C, are selected at random. A new input I' is created according to the following method, where we let A_i denote the value of the ith parameter in the input A and, likewise, for the other inputs: For each parameter value I_i in the input I, we let $I'_i = I_i$ with probability p, where p is a parameter to the genetic algorithm. With probability $1 - p$, we let $I'_i = A_i + \alpha(B_i - C_i)$, where α is a second parameter of the GA. If I' results in a better objective function value than I, then I' replaces I; otherwise, I is kept. This procedure can be thought of as an operation on ℓ-dimensional vectors. First, we generate a new vector by adding a weighted difference of B and C to A. Then, we perform ℓ-point crossover between A and the newly generated vector, obtaining our result. This is illustrated in Fig. 3. Here, the random selection has come out in such a way that the first, third, and fourth elements of A, namely, A_1, A_3, and A_4, are copied directly to the result; perhaps p was about $2/3$. But, the second element of the result has the value $A_2 + 0.4(B_2 - C_2)$ and the fifth element has the value $A_5 + 0.4(B_5 - C_5)$.

3.3 An Example of Genetic Search in Dynamic Test Data Generation

Our example of test generation by genetic search is based on the following simple program:

```
main(int argc, char **argv)
{
    int a = atoi(argv[1]);

    if (a > 100)
      puts("hello world\n");
}
```

The program input is read in by the statement `int a = atoi(argv[1])`, which takes the first argument from the command line and assigns it to the variable `a`. (Although the test generator only generates scalar input values, those values are represented as text when the program needs them in that format). Suppose the test requirement is simply to exercise the `puts` statement on line 6, causing "`hello world`" to be printed.

The objective function will be based on the value of variable `a` at line 3 since this variable determines whether or not line 4 is reached. The source code is instrumented in a way that causes the value `a - 100` to be sent to the test generator when line 3 is executed. This is the value of the objective function $\mathfrak{F}$.

When the genetic algorithm is invoked, it begins by generating an initial population of test inputs; each input will be treated as one individual by the genetic algorithm. For this example, we assume four individuals are generated with the values 94, 91, 49, and -112, respectively.

The value returned during the four test executions is used to determine the fitness of each of the four inputs. A smaller number indicates that the test input is closer to satisfying the criterion, but, in genetic search, *larger* numbers have traditionally been associated with greater fitness and we will adopt that convention here. Therefore, the fitness of each input is obtained by taking the inverse of the heuristic value returned when the target program is executed. For the inputs -112, 49, 91, and 94, the instrumented program returns 222, 51, 9, and 6, respectively, to the execution manager. Therefore, the respective fitnesses of the four inputs are $1/222$, $1/51$, $1/9$, and $1/6$.

Once the fitness of each test input has been evaluated, the reproduction phase of the genetic algorithm begins. Two inputs are selected for reproduction according to their fitness. This is accomplished by normalizing the fitness values and using them as the respective probabilities of choosing each input as a parent during reproduction. The probabilities of selecting -112, 49, 91, and 94 are thus 0.015, 0.065, 0.368, and 0.552, respectively. With only four individuals, there is a significant probability of selecting the same individual twice, but many genetic algorithms have explicit mechanisms that prevent this. We will assume the algorithm in our example does so as well.

Suppose the two parents selected are 91 and 94—the two inputs with the largest probabilities. Reproduction is accomplished by representing each number in binary form, 01011011 and 01100000, respectively. A crossover point is selected at random (suppose it is 3), and crossover is performed by exchanging the first three bits of the two parents. The resulting offspring are 01000000 and 01111011, or 64 and 123.

Since reproduction continues until the number of offspring is the same as the original size of the population, two more inputs are selected as parents. Suppose they are -112 and 91 and the crossover point is 5. If the inputs are converted to bit-strings using 8-bit, two's complement representation, then they are 10010000 and 01011011, respectively. Crossover produces the offspring 10010011 and 01011000 or -109 and 88.

In summary, the second generation consists of the individuals 64, 123, -109, and 88. When the program is executed on these inputs, it is found that one of them satisfies the test requirement, so the test generation process is complete (for that requirement). If none of the tests had met the requirement, the reproduction phase would have begun again using the four newly generated inputs as parents. The cycle would have been repeated until the test requirement was satisfied or until the GA halted due to insufficient progress.

(Note that we would have been in trouble if all four original test inputs had had zeros in the first two binary digits. Then, no combination of crossovers could have created a satisfactory test input and we would have had to wait for an unlikely mutation. This is part of the importance of diversity, mentioned in Section 5.1 when we discuss the adjustment of the GA's parameters.)

3.4 Other Optimization Algorithms

Once the underlying framework for dynamic test data generation has been implemented, it is straightforward to add other optimization techniques. One such technique is simple gradient descent.

We implemented two gradient descent algorithms: Polak-Ribiere conjugate gradient descent [28] and a reference algorithm which is slow but makes no assumptions about the shape of the objective function.

Conjugate gradient descent is best suited for optimization problems with continuous parameters, and integer parameters lead to a myriad of small plateaus where the algorithm can stall. That is, the algorithm may make such small adjustments to an integer-valued parameter that there is no affect on the program's behavior. (Continuous-valued parameters from the optimization algorithm were converted to integers where necessary by rounding). To solve this problem, we interpolate the objective function, but our technique seeks to execute the program as infrequently as possible and is somewhat crude in other respects. For the input values $x_1, x_2, \cdots, x_n$ found by the gradient descent algorithm, we calculate the two values

$$\mathcal{G}_1 = \Im(y_1, y_2, \cdots, y_n)$$

and

$$\mathcal{G}_2 = \Im(z_1, z_2, \cdots, z_n),$$

where

$$y_i = \begin{cases} x_i, & \text{if } x_i \text{ is a continuous parameter;} \\ \mathbf{rnd}(\mathbf{x_i}), & \text{if } x_i \text{ is an integer parameter} \end{cases}$$

and

$$z_i = \begin{cases} x_i, & \text{if } x_i \text{ is a continuous parameter;} \\ \mathbf{rnd}(\mathbf{x_i}) + \mathbf{1}, & \text{if } x_i \text{ is an integer parameter and} \\ & x_i > \mathbf{rnd}(\mathbf{x_i}); \\ \mathbf{rnd}(\mathbf{x_i}) - \mathbf{1}, & \text{if } x_i \text{ is an integer parameter and} \\ & x_i < \mathbf{rnd}(\mathbf{x_i}), \end{cases}$$

where **rnd** denotes rounding to the nearest integer. (Our implementation rounds 0.5 down.)

The interpolated value is then

$$\mathcal{G}_1 - \frac{d_E(y_1, y_2, \cdots, y_n; x_1, x_2, \cdots, x_n)}{d_E(z_1, z_2, \cdots, z_n; x_1, x_2, \cdots, x_n)}(\mathcal{G}_1 - \mathcal{G}_2),$$

where $d_E(y_1, y_2, \cdots, y_n; x_1, x_2, \cdots, x_n)$ denotes the Euclidean distance between the point $y_1, y_2, \cdots, y_n$ and the point $x_1, x_2, \cdots, x_n$.

Once a program input is found that satisfies the current test criterion, the gradient descent procedure is terminated. This can happen before the procedure believes that it has reached the true minimum, both because of the way we define the objective function and because of distortions caused by the interpolation technique.

Conjugate gradient descent is intended to be used when the objective function is roughly quadratic, and this may not be the case during test data generation. Furthermore, our interpolation technique might be poorly suited to the problem and lead to less than ideal results.

To evaluate the impact of potential problems, we also implemented a brute-force gradient descent algorithm for reference. The reference algorithm makes few assumptions about the form of the objective function, although it is slow.

This algorithm is quite simple. It begins with a seed input and measures the objective function for all inputs in the neighborhood of the seed. The neighboring input that results in the best objective function value becomes the new seed, and the process continues until a solution is found or until no further improvement in the objective function seems possible.

A number of techniques can be used to define a *neighborhood function*, which determines what the neighbors of the seed are. In our experiments, the neighborhood was formed by incrementing and decrementing each input parameter, so that an input containing 10 parameters (creating a 10-dimensional input space) has 20 neighbors. GADGET also allows the dimensionality of the input space to be adjusted by treating the input string as a series of bitfields, which are treated as integers when they are incremented or decremented during gradient descent.

The algorithm permits random selection of the step size, which determines how much an input value is incremented or decremented. For most of our experiments, the step sizes were chosen according to a Gaussian distribution whose mean was a user-supplied parameter and whose standard deviation was half of the mean.

To help prevent the algorithm from stepping over a solution, the mean step size is halved when all neighbors

have a higher objective-function value than the seed. In this way, the algorithm implements a technique, also used by conjugate gradient descent, of bracketing a minimum in the objective function and then conducting an increasingly refined search for the minimum within the bracket.

4 The Genetic Algorithm Data GEneration Tool (GADGET)

In this section, we describe our approach to test data generation by genetic search. Recall from Section 2.2.3 that our approach is based on the conversion of the test-data generation problem to a function minimization problem, which allows a genetic algorithm to be applied. In the previous section, we described the genetic algorithm itself, and we now describe its application to the problem of automatic test data generation. We begin with an overview of the algorithm, and then we discuss a number of issues in greater detail, including the implementation of the fitness function and the technique for ensuring that the GA can reach the location where the fitness function is evaluated.

4.1 Overview of the Test Data Generation Algorithm

The goal for the GAs is to find a set of tests that satisfy condition-decision coverage (see Section 2.1). This leads to two test requirements for every condition in the code, namely, that the condition be true for at least one input and false for at least one input. Condition-decision coverage also requires that each branch of each decision be taken at least once, but this requirement is satisfied by any test set that meets the requirements above.

Before starting the GA, we execute the program on a seed input. Such a seed input typically satisfies many of the test requirements (see Section 5.2 for a discussion of this). The initial program execution is used to initialize the coverage table. After this, the coverage table is used to select a series of test requirements in turn. For each test requirement, the GA is initialized and attempts to satisfy the given requirement. Due to the limitation on the number of iterations (the GA must make some progress every n iterations for some n), the GA is guaranteed to halt, either because a solution has been found or because the GA has given up.

Whenever the GA generates an input that satisifes a new test requirement, whether or not that test requirement is the one the GA is currently working on, the new test input is recorded for future use (see below), and the coverage table is updated. We also record how many times the target program was executed before the new input was found.

The test generation system continues to iterate over the test requirements until no further progress can be made. This happens when one attempt has been made to satisfy every reachable requirement. (A requirement is considered unreachable if no test input executes the code location where the requirement is evaluated. In other words, the test requirement refers to a condition that the GA cannot reach. The coverage table can be used to determine when there are no more reachable requirements, see Section 4.2.)

The performance of each test generator is measured as the percentage of test requirements it has satisfied. To determine this percentage, the inputs that are found by the test generator are evaluated by a commercial coverage analysis tool (DeepCover), together with the original seed input. This lets us plot the percentage of requirements satisfied as a function of the number of times the target program has been executed.

The two details that must be addressed are the same as those described in Section 2.2.3: We must find a way to reach the code location where we want our test requirement to be satisfied, and we must convert that requirement into a function that can be minimized.

4.2 Reaching the Target Condition

Recall that, in dynamic test generation, function minimization cannot be performed unless the flow of control reaches a certain point in the code. For example, if we are seeking an input that exercises the TRUE branch of a condition in line 954 of a program, we need inputs that reach line 954 before we can begin to do function minimization.

Our approach is slightly different from that of [2] and [5], which concentrate on finding a specific path to the desired location. Our goal is (among other things) to cover *all* branches in a program. This means we can simply delay our attempts to satisfy a certain condition until we have found tests that reach that condition.

This leads to a test generation approach similar to the one employed by [21]. A table is generated to keep track of the conditional branches already covered by existing test cases. If neither branch of a condition has been taken, then that decision has not been reached, so we are not ready to apply function minimization to that condition. If both branches have been taken, then coverage is satisfied for that condition and we need not examine it further. However, if only one branch of a condition has been exercised, then the condition has been reached, and it is appropriate to apply function minimization in search of an input that will exercise the other branch.

4.3 Calculation of Fitness Functions.

Recall from Section 2.2.3 that dynamic test generation involves reducing the test generation problem to one of minimizing a fitness function $\mathfrak{F}$. The first step is to define $\mathfrak{F}$. Like other dynamic test generation techniques, ours begins by instrumenting the code under test. The purpose of this instrumentation is to allow us to calculate the fitness function by executing the instrumented code.

At each condition in the code, our system adds instrumentation to report $\mathfrak{F}(x)$ when execution reaches that condition. Table 2 shows how $\mathfrak{F}(x)$ is calculated for some typical relational operators when we are seeking to take the TRUE branch of a condition (the functions for the FALSE branch are analogous).

If the program's execution fails to reach the desired location (it terminates or times out without having executed the statement), then the fitness function takes its worst possible value.

Since our system seeks to generate condition-decision adequate test sets, conjunctions and disjunctions can be handled by exploiting C/C++ short circuit evaluation. For example, the second clause of the condition

```
if ( (c > d) && (c < f) ) ...
```

TABLE 2
Computation of the Fitness Function

decision type	*example*	*fitness function*
inequality	`if (c >= d)...`	$\mathfrak{F}(x) = \begin{cases} \text{d-c}, & \text{if } d \geq c; \\ 0, & \text{otherwise} \end{cases}$
equality	`if (c == d) ...`	$\mathfrak{F}(x) = \lvert d - c \rvert$
true/false value	`if (c) ...`	$\mathfrak{F}(x) = \begin{cases} 1000, & \text{if } c = \text{FALSE}; \\ 0, & \text{otherwise} \end{cases}$

is not reached unless the first clause evaluates to TRUE, so the requirement that states the first and second clause must both be TRUE is replaced by a requirement stating that the second clause must be reached and must evaluate to TRUE. If both clauses are reached and both clauses take on both the value TRUE and the value FALSE (as required by condition-decision coverage), then both branches of the conditional branch will necessarily have been taken.

4.4 Execution Control

In the GADGET system, an execution controller is in charge of running the instrumented code, coordinating GA searches, and collecting coverage results and new test cases. It begins by executing all preliminary test cases. (These preliminary cases can be supplied by the user or generated randomly by the execution controller.) After running all initial test cases, the execution controller uses the coverage table to find a condition that can be reached, but has not been covered yet (that is, no input has made the condition TRUE or else no input has made it FALSE). The genetic algorithm is invoked to make this condition take on the value that was not already observed. The GA is seeded with test cases that can successfully reach the condition (though they did not give the condition the desired value, or else the condition would already have been covered). If additional inputs are needed to get the required population size, they are generated randomly.

When the GA terminates, either because it found a successful test case or because it stopped making progress, the execution controller uses the coverage table to find a new condition that has not been covered completely. The GA is called again with the task of finding an input that covers this condition. This process continues until all conditions that have had only one value (either TRUE or FALSE) have been subjected to GA search. The execution controller keeps track of all GA-generated test inputs that cover new program code, regardless of whether or not they satisfy the test requirement that the GA is currently working on. (In other words, GADGET takes advantage of all serendipitous coverages.) These test cases are stored for later use.

4.5 Computational Costs

Since dynamic test generation is a hueristic process, we cannot give a universal characterization of its computational cost. To be specific, the optimization algorithm makes iterative improvements to a test input while trying to make that input satisfy the chosen requirements, and we cannot predict exactly how many iterations there will be.

However, there are a number of factors that influence the computational cost of each iteration. The most significant cost is that of executing the target program; recall that this program has to be executed in order to evaluate the objective function. Each new generation created by the genetic algorithm contains a number of new test inputs, and the objective function has to be evaluated for each one. Thus, the cost of dynamic test generation depends intimately on the cost of executing the target program. In addition, larger programs typically have more conditional statements, so if we try to obtain *complete* coverage of a program, as we did in the experiments described in Section 5, the number of different optimizations the GA has to perform depends on the program size as well.

The second factor that determines the efficiency of test generation is the optimization algorithm itself. For example, the genetic algorithm's ability to escape local minima comes at the cost of additional resource expenditures, compared to gradient descent. In other words, the genetic algorithm makes more iterations than gradient descent would make, and it executes the target program more often.

The experiments we report on in Sections 5.3 and 5.4 required anywhere from 30 minutes to several hours on a Sun Sparc-10 workstation, though the simple programs in Section 5.2 were less expensive. Based on results reported in [4] and elsewhere, this genetic search may be considerably more expensive than gradient descent would be (but see below). Ultimately, such expense would be justified by the difficulty of generating test data by hand, which makes any automated technique desirable, especially for programs with a complex structure.

It is interesting to note that computational overhead seems to play a considerable role in the time needed to perform dynamic test generation. When running the target program on a given test input, the test generator has to invoke the program (using the Unix `vfork` and `exec` system calls in the case of GADGET) and wait for the results (GADGET uses `wait`.) For all of our programs, even the largest, computation time was significantly affected by the expense of the Unix `vfork/exec/wait` calls needed to execute the target program. We empirically estimated that our operating system is capable of about $3,200$ `vfork/exec/wait` operations per minute. In contrast, [4] reports being able to perform $200,000$ to $700,000$ tests in five minutes (using random test data generation). Further computational overhead comes from the fact that GADGET is written in object-oriented C++ and was not optimized for these experiments. This (together with the timing results of [4]) suggests that it is possible to obtain much more efficient operation.

Apparently, platform-dependent computational overhead greatly affects the performance of test-generation algorithms. Therefore, it seems that the number of executions of the target program is a better measure of a test-generator's resource requirements than wall-clock or system time. Reference [4] does not report on the number of executions used by their nonrandom techniques and, although it appears that their gradient descent technique was much cheaper than our genetic search, their reduced overhead costs call this conclusion into question, at least if computational expense is measured in terms of the number of executions of the target program.

In the next section, we do, indeed, measure the expense of test generation by the number of target-program executions. For reasons discussed above, this seems to be the most logical measure of efficiency for a dynamic test generator: It abstracts away the cost of actually running the target program itself, as well as the overhead of calling the target program and communicating with it.

4.6 Tuning the GA

When a test is performed using one of the two genetic algorithms described in Section 3.2, the GA is first tuned by adjusting the number of individuals in the population and the number of iterations that must elapse with no progress before the GA gives up. In the standard GA, mutation is controlled by adjusting the probability that any single bit will be flipped during reproduction. The goal of this fine-tuning was to maximize the percentage of conditions covered while keeping the execution time low. A second goal was to ensure that the differential and standard GAs executed the target program about the same number of times in order to get a reasonable comparison between the two techniques.

Such tuning adjustments control the breadth and the thoroughness of the GA's search. If there are more individuals, then there are more disinct inputs that can be created by the crossover operation; in a sense, more genetic diversity is available. On the other hand, if we make the GA continue for more iterations before giving up, then it is less likely to give up simply because progress is slow. In some cases, the GA is visibly on the path to a successful input, but it gives up because an unlucky series of crossovers fails to improve the fitness of the population. The likelihood of this is reduced if more iterations are permitted.

Unfortunately, both of these improvements have a cost. Allowing more iterations means the GA will waste more time trying to cover conditions that it cannot cover. Adding more individuals means that the target program is executed more often during each iteration because a target-program execution is required every time we evaluate the fitness of an individual.

5 Experimental Results

In this section, we report on four sets of test data generation experiments. The first set of experiments involves programs that calculate simple numeric functions. The second experiment investigates how the GAs and random test generation perform on increasingly complex synthetic programs. Finally, we present results obtained by analyzing a real-world autopilot control program called `b737`.

5.1 Design of the Experiments

In setting up our experiments, we began by selecting a program on which to try out the test generation system; we refer to such a program as a *target program*. By means of source-code instrumentation, the target program is configured to run in tandem with an external execution manager, which monitors the program's behavior. The target program is instrumented by augmenting each condition with additional code. This code reports the objective-function value for each condition to the execution manager. (The objective functions are calculated as described in Section 4.3). The program that performs this instrumentation also collects the information needed to construct the coverage table described in Section 4.2.

Several parameters of the GA were the same throughout all of our experiments. For the differential GA, the parameter p (see the end of Section 3.2) was always 0.6. The parameter α was always 0.5. The probability of mutation, number of individuals, and number of iterations permitted without progress were different for different test programs. We report the results for each of the programs we tested and give the actual values we used below. (See Section 4.6 for a discussion of how these parameters affect the performance of the GA.)

An additional parameter of the test generator is the random number seed used to create pseudorandom numbers by means of a linear congruential generator (c.f., [34]). The following describes the utilization of pseudorandom numbers and, hence, the impact of the choice of a random number seed in each of the three test generators.

- **Random test generation.** All inputs are generated pseudorandomly. In all of our experiments, the parameters were numeric, and inputs were generated by selecting a value for each parameter uniformly at random from the range specified in a program-specific configuration file.
- **Gradient descent.** The initial inputs are randomly generated using the same technique as the random test data generator. Since our implementation supports it readily, we randomly select two starting points and perform the two descents in tandem with one another. The step size has a pseudorandom element as well, as outlined earlier at the end of Section 3.4.
- **Standard genetic search.** The seed inputs are created pseudorandomly, as are new population members when these are needed. Each input is represented as a string of bits, and inputs are generated by setting each bit to 1 with probability 0.5. Crossover points are selected uniformly at random from the potential crossover points in the bit-string, and the decision of whether or not to mutate a bit is based on a randomly generated number as well.
- **Differential genetic search.** Pseudorandom numbers are used to create the seed input and to create new population members as above and, in addition, the random number generator drives the random choices used in reproduction (see Section 3.2).

We executed the test generation system described in Section 4 using each of the genetic algorithms described in Section 3.2. A single run of the test generation system involves the following steps:

1. Selection of a target program for which tests will be generated.
2. Specification of a test adequacy criterion. In these experiments, the test adequacy criterion is always condition-decision coverage.
3. Enumeration of the test requirements that the adequacy criterion induces on the target program.
4. An attempt by the test generator to satisfy *all* of the test requirements for the program.

We performed several complete test-generation runs with each test generation system and for each of the several test programs. (That is, each test generation system made several attempts to satisfy the test requirements associated with each program.) The exact number of runs varies between test programs, and it is given below when we describe our results for each of the programs. During each run of the test generation system, the system saves the original seed input, and it also saves any test input that satisfied a requirement not already satisfied by an earlier input.

After each run of the test generator, information collected during the run is used to assess the test generator's performance. Performance is measured in terms of the percentage of test requirements that were satisfied. To do this, the seed input is fed to a commercial coverage analysis tool (DeepCover), along with the inputs that were saved by the test generator because they satisfied new coverage requirements. This tells us how many requirements were satisfied by these inputs.

Each time the test generator saves a test input, it records how many times the target program was executed before that input was found. This lets us plot the test generator's performance as a function of the number of target-program executions, which is what we do in most of our experiments.

In addition to running the two GA-based test generators, we also apply the random test generator to each target program. We run the random test generator as many times as the two other algorithms. For each run of the random test generator, we execute the program on randomly generated input values and record the inputs that satisfy new requirements as above. The number of such executions is equal to the largest number of times the program was executed by any of the GAs. We plot the performance of the random test generator in the same way as the performance of the GAs.

5.1.1 Input Representation in the GAs

Input representation in the GAs. In the standard GA, test inputs are represented as a contiguous string of bits, and crossover is accomplished by selecting a random position within the bit-string. (In our experiments, the binary representation was obtained by using the machine encoding for floating-point parameters and Gray-coding others.) Representing the input as a series of bits does not work for the differential GA (see Section 3.2) because its method of reproduction is not well-suited to variables with only two values. Therefore, the differential GA sees each input as a series of variables corresponding to the input parameters of the target program. The description of the input parameters (ranges, types, etc.) is provided to the test generation system by the user and that description is used both by the standard GA and the differential GA when test inputs are formatted for use by the target program.

5.2 Simple Programs

We began our experiments on a set of simple functions much like those reported in the literature [35], [21], [4]. The programs analyzed are as follows:

- binary search,
- bubble sort,
- number of days between two dates,
- euclidean greatest common denominator,
- insertion sort,
- computing the median,
- quadratic formula,
- warshall's algorithm, and
- triangle classification.

These programs are roughly of the same complexity, averaging 30 lines of code and all having relatively simple decisions.

When we tested GADGET on these programs, we used 30 individuals for the differential GA and allowed 10 generations to elapse with no progress before allowing the GA to give up. For the standard GA, we used 100 individuals, allowed 15 generations to elapse before the GA gave up, and made the probability of mutation 0.001.

For these programs, random test data generation never outperforms genetic search, though sometimes both approaches have the same effectiveness. These results resemble those reported in [21] and [4]; in those papers, random test generation also performed nearly as well as more sophisticated techniques on simple programs.

Random test case generation has the upper hand in these experiments because it involves significantly less computation. However, in every case, one of the GAs performs the best overall. Table 3 shows the results. The numbers reported in the table represent the highest percentage of test requirements satisfied by any single run of the test generator among a series of five such runs.

It is useful to analyze a sample program to understand what the GAs are doing. The code for **Triangle classification** is shown below. (The comments on the right margin will be used later to show which inputs satisfied which conditions.) Note that many decisions only contain a single condition.

```
#include <stdio.h>

int triang (int i, int j, int k) {
  // returns one of the following:
  // 1 if triangle is scalene
  // 2 if triangle is isosceles
  // 3 if triangle is equilateral
  // 4 if not a triangle

  if ( (i <= 0) || (j <= 0) || (k <= 0) ) return 4;
    // acd
```

TABLE 3
A Comparison of Four Test Data Generation Approaches on Simple Programs

Program	random	GD	GA	differential-GA
Binary search	80	100	70	100
Bubble sort 1	100	100	100	100
Bubble sort 2	100	100	100	100
Number of days between two dates	87.5	90	100	100
Euclidean greatest common denominator	100	100	100	100
Insertion sort	100	100	92.9	100
Computing the median	100	90	100	100
Quadratic formula	75	75	75	75
Warshall's algorithm	91.7	100	100	100
Triangle classification	48.6	91.7	94.29	84.3

The standard GA exhibits the best performance overall, but not by a significant amount. These data represent the highest percentage coverage obtained by each method during a series of five attempts to obtain complete coverage of the program.

```
  int tri = 0;

  if (i == j) tri += 1;        // g
  if (i == k) tri += 2;        // h
  if (j == k) tri += 3;        // i
  if (tri == 0) {              // bef
      if ( (i+j <= k) || (j+k <= i) || (i+k <= j) )
      tri = 4; // be
      else tri = 1;        // f
      return tri;
      }
  if      (tri > 3) tri = 3;
  else if ( (tri == 1) && (i+j > k) ) tri = 2;
  else if ( (tri == 2) && (i+k > j) ) tri = 2; // h
  else if ( (tri == 3)  (j+k > i) ) tri = 2;
  else tri = 4;
  return tri;
  }

void main() {
  printf("enter 3 integers for sides of
  triangles\n");
  int a,b,c;
  scanf("%d %d %d",&a, &b, &c);
  int t = triang(a,b,c);
  if (t == 1)       printf("triangle is
                    scalene\n");     // f
  else if (t == 2)  printf("triangle is
                    isosceles\n");   // h
  else if (t == 3)  printf("triangle is
                    equilateral\n");
  else if ( t== 4)  printf("this is not a
                    triangle\n");    // abcdegi
```

Fig. 4 shows how the four different systems—standard GA, differential GA, gradient descent, and random test generation—perform on the **triangle** program shown above.

5.2.1 Coverage Plots for Test Generation Problems

In Fig. 4, the number of test requirements satisfied by each test generation system is plotted against the number of executions of the target program. Here, as well as in our later results, the standard GA and differential GA did not execute the program the same number of times. (In general, the standard GA tended to satisfy individual test requirements more quickly.) The random test generator was run last, and the number of program executions alloted to it was the maximum number of program executions needed by either of the other two systems. This creates favorable conditions for the random test generator and helps us determine when the use of a more complex test generation system is actually justified.

The plot in Fig. 4 shows features that appear throughout our test generation experiments. First, there is a large, immediate jump in the percentage of test requirements

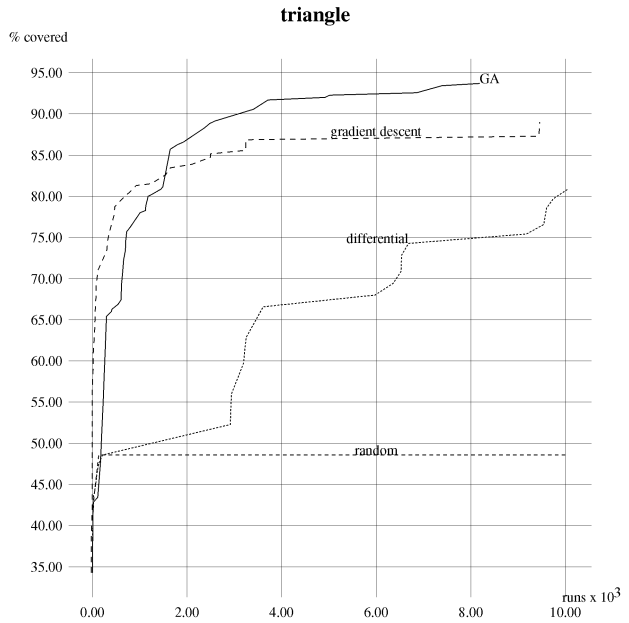


Fig. 4. A coverage plot comparing the performance of the two GAs, gradient descent, and random test generation on the **triangle** program. The graph shows how performance (in terms of the percentage of the 20 test requirements that have been satisfied) improves as a function of the number times the program is executed. Random test generation hits its peak early, but fails to improve after that. The differential GA has a better performance and executes for a longer amount of time, but the standard GA has the best performance overall, covering about 93 percent of the code on average in about 8,000 target-program executions. Gradient descent performs nearly as well as the standard GA. The curve for each system represents the pointwise mean performance of that system taken over five attempts by that system to achieve complete condition-decision coverage of the program.

TABLE 4
A Table of Sample Input Cases Generated by the Standard GA for **triangle**

Execution of target program	Key	Integer1	Integer2	Integer3
2	a	1680498885	1961702355	-1490056820
3	b	1293470477	1898197634	465181194
4	c	-120192928	1041962067	280365949
6	d	841354299	-1802686561	-209782592
20	e	1056804119	660913846	1617709752
117	f	719320455	507534636	574028437
5311	g	743820356	743820356	1826109949
10751	h	999699718	584551117	999699718
16800	i	799340978	1321708382	1321708382

These data can be mapped to conditional expressions in the code shown above using the Key field.

satisfied, so that the interesting part of the vertical axis does not start at zero but somewhere closer to 50 percent. Second, the percentage of requirements satisfied by the GAs seems to make discontinuous jumps.

The initial jump in coverage is there because the first test input satisfies many conditions. When the program executes the first time, it has to take either the TRUE branch or the FALSE branch on any condition it encounters and, each time a new condition is found to be true or false, the percentage of test requirements satisfied goes up. For example, if a program has no nested decisions and if each decision has exactly one condition, then the first program execution is forced to take either the TRUE branch or the FALSE branch of every condition in the program. For such a program, we would always achieve 50 percent coverage using only the first test.

Many of the discontinuities that appear in Fig. 4 are there for similar reasons. When an input causes the program to take a branch that had not been taken before, it may lead to the execution of other conditional statements that were not executed previously. Once again, each condition must either be TRUE or FALSE, and this leads to an instantaneous increase in the number of conditions satisfied. (Apparent discontinuities can also occur when there are few conditions in the program because then the granularity of the vertical axis is low. However, this is not the cause of the salient jumps in the coverage plots we present here. Readers may visually judge the granularity of the plots by looking at small features, such as the shallow increments that occur in the plot for the standard GA at 43 percent and 78 percent on the vertical axis.)

A further cause of discontinuities is serendipitous coverage. The GAs often satisfied one test requirement by coincidence when they were trying to satisfy a different requirement. In fact, the shorter execution time of the standard GA results largely from this phenomenon; less exertion was required of the GA because so many requirements were met serendipitously. We find this to be a recurring phenomenon in our experiments and have more to say about it in Section 5.5.4.

5.2.2 The GAs' Choices of Test Inputs

A sample of results obtained by the GA test data generation algorithm is shown in Table 4. These data can be mapped to the source code shown above by using the letters shown in the comments.

The tests in Table 4 are probably not like those that a human tester would choose, much less those that would occur in a hypothetical world where this program was used by the general public. Of course, the ability to create bizarre tests can sometimes be an advantage. Coverage criteria are often used to exercise obscure and infrequently utilized features of a program, which testers might otherwise overlook. However, it might be desirable to concentrate on inputs that are more realistic. This leads to an interesting digression on dynamic test generation techniques: Unlike the static methods described in Section 2.2.2, dynamic test generation allows certain inputs to be preferred over others by means of biases in the objective function. For example, bizarre or uncommon input combinations could be penalized by raising the value of the objective function for those inputs. One could even construct an operational profile (c.f., [36]), allowing each input to be weighted according to its probability of occurring on the field. Similar biases are used in [22] to create a preference for test inputs close to boundary values.

5.2.3 Performance of Gradient Descent

In view of the above discussion, it is perhaps unsurprising that the performance of gradient descent was generally somewhere between that of the genetic algorithms and that of random test generation. In many cases, gradient descent failed because of flat spots in the objective function where there is no information to guide the algorithm's search. This was the case with the binary search program, for example.

However, gradient descent appears to encounter a local minimum in the `triangle` program. This can be observed in the behavior of the reference algorithm for gradient descent, which empirically estimates the gradient at each step of the optimization process. In the `triangle` program, the reference algorithm reaches a point where all adjacent points (as defined by the neighborhood function) lead to a worsening of the objective-function value. This leads to a reduction of the mean step size (as described in Section 3.4), but the same phenomenon is encountered again, and this process is repeated until the algorithm gives up. The optimization process is empirical rather than analytic, so this does not prove the existence of a local minimum, but it provides strong evidence. (Of course, this might be prevented by a clever neighborhood function, but finding a neighborhood function that is guaranteed to eliminate local minima is a nontrivial matter.)

The test criterion whose objective function contained the local minimum was satisfied by the standard GA, albeit serendipitously.

5.2.4 Performance of the Random Test Generator

Our results for random test case generation resemble those reported elsewhere for cases where the code being analyzed was relatively simple. For the simple programs in Table 3, the different test generation techniques have roughly the same performance most of the time. In [21], random test data generation was reported to cover 93.4 percent of the conditions on average. Although its worst performance—on a program containing 11 decision points—was 45.5 percent, it outperformed most of the other test generation schemes that were tried. Only symbolic execution had better performance for these programs. Reference [4] reports on 11 programs averaging just over 100 lines of code. Overall, random test data generation was fairly successful, achieving 100 percent statement coverage for five programs and averaging 76 percent coverage on the other six.

It is also interesting to compare our results with those obtained by [16] for three slightly larger programs. Again, simple branch coverage was the goal. Random test generation achieved 67 percent, 68 percent, and 79 percent coverage, respectively, on the three programs analyzed. Symbolic test generation achieved 63 percent, 60 percent, and 90 percent coverage, while dynamic test generation achieved 100 percent, 99 percent, and 100 percent coverage.

These results suggest a common trend: Random test generation has at least an adequate performance on such programs, but, for larger programs or more demanding coverage criteria, its performance deteriorates. The programs used in [16] were larger than those used in [21] and random test generation had poorer performance on the larger programs.

The results reported in this section also use small programs, though our test-adequacy criterion is more stringent. In some cases (like the triangle program), random test generation is far less adequate, even though, in our experiments, resource limitations were not a telling factor: The final 90 percent of the randomly generated tests failed to satisfy any new test requirements. In our subsequent experiments on larger programs, we find this to be a continuing trend: Program size and program complexity decreased the percentage of test requirements that could be satisfied using random test generation.

In our experiments, the percentage of test requirements satisfied by random test generation actually goes down when the programs become more complex. This suggests that something about large programs, other than the sheer number of conditions that must be covered, makes it difficult to generate test data for them. This is also suggested in our later experiments.

5.3 The Role of Complexity

In our second set of experiments, we created synthetic programs with conditions and decisions whose characteristics we controlled. The two characteristics we were interested in controlling were: 1) how deeply conditional clauses were nested (we call this the *nesting complexity*) and 2) the number of Boolean conditions in each decision (which we call the *condition complexity*). For example, a program with no nested conditional clauses (nesting complexity = 0) would look like the beginning part of the **triangle** program, in that `if` statements are not nested inside other `if` statements. The nature of the conditional expressions in each conditional clause is controlled by the second parameter (the condition complexity). The decision `(( i<= 0) || (j <= 0) || (k <= 0))` from the **triangle** program ranks as a 3 on this scale because it contains three conditions. In this section, we will use these two parameters to characterize the complexity of our synthetic programs. We will use the expression *compl(x, y)* as shorthand for "nesting complexity x and condition complexity y." In our experiments, programs were generated with all complexities $compl(nest, cond)$, $nest \in \{0, 3, 5\}$, $cond \in \{1, 2, 3\}$. In this section, we present the results for $compl(0, 1)$, $compl(1, 3)$, and $compl(3, 5)$, which illustrate the widening gap between the performance levels of the three test generators as program complexity grows. The results for the other six programs are given in the appendix.

Note that, in some cases, changing the nesting complexity has the same effect as changing the condition complexity. Changing `if (cond1 && cond2)` to `if (cond1) if (cond2)` does not change the semantics of a program in `C` or `C++`, but it does change the nesting and condition complexities. However, in our synthetic programs, there is always additional code between two nested conditions. Thus, a higher nesting complexity implies a more complicated relationship between the input parameters and the variables that appear in deeply nested conditions. On the other hand, a high condition complexity implies that many conditions are evaluated in the same program location, meaning that the test generator has to find a single set of variable values that satisfies many simple conditions at the same time.

In these tests, the differential GA had a population size of 30 and abandoned a search if no progress was made in 10 generations. The standard GA also gave up after 10 generations made no progress, but the population size was adjusted until the standard GA executed the target program about the same number of times as the differential GA. This resulted in a population size of 270 for $compl(0, 1)$, 320 for $compl(3, 2)$, and 340 for $compl(5, 3)$. The large population sizes make up for the standard GA's tendency to give up early; this is discussed in Section 5.3.1.

The mutation probability for the standard GA was 0.001.

Figs. 5, 6, and 7 show the mean performance over six runs of each test generation technique. (Note that the percentage of test requirements satisfied, the shown vertical axis, has different ranges in different plots. We have focused on the interesting portion of each plot.)

For the simplest program, random test generation and the GAs quickly achieve high coverage. All three methods make most or all progress during the very early stages of test generation. In these early stages, the GAs have essentially random behavior because evolution has not begun yet. The standard GA sometimes satisfied additional requirements closer to the end of the process. The results are shown in Fig. 5, whose horizontal scale is logarithmic to show both of the features just mentioned.

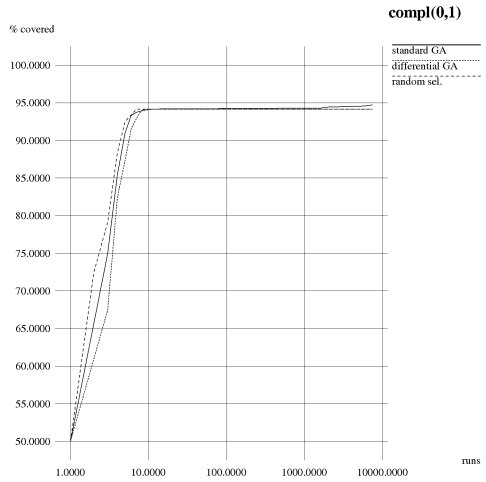


Fig. 5. The random test generator and GAs perform relatively equally on the least complex program. $compl(0,1)$; 100 percent coverage represents coverage of 60 conditions.

Fig. 6 shows results for a program of intermediate complexity. Random test generation does not do as well as before. When we examined the details of each execution, we found that the standard GA performed well because it often managed to satisfy new criteria serendipitously—that is, it often found an input satisfying one criterion while it was searching for an input that satisfied another. It is likely that the differential GA performs a more focused search than the standard GA, and our examination of the detailed results showed that it failed to find as many inputs by coincidence.

Finally, Fig. 7 shows the results for a program with $compl(5,3)$. Here, all test generation methods are less effective than before, and there is more to distinguish among the different techniques.

In most cases, the standard GA performed better than the differential GA.

The results for conjugate gradient descent and the reference algorithm are shown in Table 5. We do not show the results for gradient descent in the coverage plots because the disparity in the horizontal scale makes them

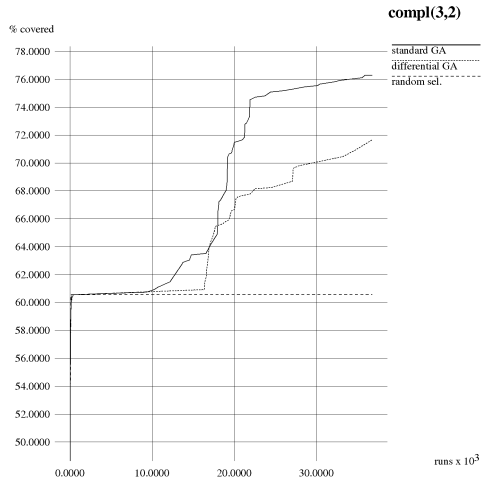


Fig. 6. As complexity increases, the GAs begins to do better and the standard GA outperforms the differential GA. $compl(3,2)$; there are a total of 45 conditions to cover.

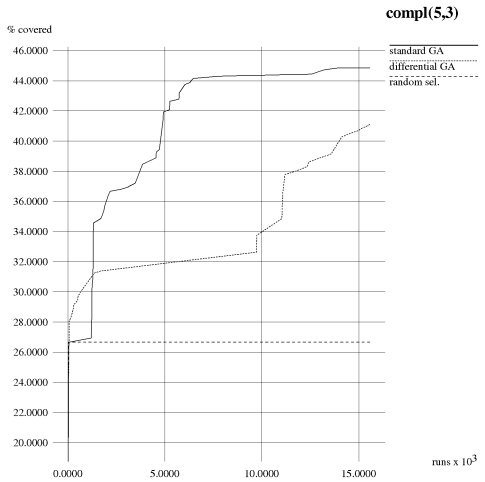


Fig. 7. The complexity program was hard for both generators, but the GA outperforms random by even more. The standard GA ultimately outperforms the other two methods. $compl(5,3)$; 45 conditions were to be covered.

hard to see; conjugate conjugate gradient descent is considerably faster than the other techniques, while the reference algorithm is somewhat slow (in any case, we expect faster performance by conjugate gradient descent a priori, so a comparison of performance over time is not as informative as it is with the two GAs). However, the table shows the total condition-decision coverage obtained by gradient descent for each of the three programs, $compl(0,1)$, $compl(3,2)$, and $compl(5,3)$. For $compl(0,1)$, the gradient descent algorithms perform comparably to the other techniques. For $compl(3,2)$, the performance of gradient descent is somewhere between that of the standard GA and that of the differential GA. For $compl(5,3)$, both GAs outperformed gradient descent.

Like the other optimization algorithms, gradient descent encountered problems because of flat regions in the objective function. This was the most frequent cause of failures to satisfy a test criterion. However, the reference algorithm apparently encountered one local minimum in the $compl(0,1)$ program, one in the $compl(3,2)$ program, and two in the $compl(5,3)$ program. (We concluded this on the basis of the same type of empirical evidence as in Section 5.2.)

For the the programs with $compl(5,3)$ or $compl(3,2)$, the GAs also failed to satisfy the test criteria for which there

TABLE 5
Condition-Decision Coverage Achieved by Conjugate Gradient Descent and the Reference Gradient Descent Algorithm for $compl(0,1)$, $compl(3,2)$, and $compl(5,3)$

compl	CGD	ref.
(0,1)	95.27	94.03
(3,2)	70.65	75.0
(5.3)	29.58	39.3

The curves are not shown since their scale is different from that of the other curves, but conjugate gradient descent was faster than the other algorithms (as expected). The two algorithms have comparable performance. Gradient descent generally performed somewhat below genetic search and comparably to differential genetic search.

were local minima in the objective function. The test criterion with the local minimum in the $compl(0,1)$ program was satisfied by the GAs some of the time, but only serendipitously. This suggests that the global optimization capability of the GAs is not what makes them perform better than gradient descent. The notion that other factors are more important than global optimization ability in determining the success of genetic search for these problems is also reinforced by our observations on serendipitous coverage.

It is interesting that conjugate gradient descent did not perform as well as the reference algorithm for most of these programs. In most cases, this is apparently because the reference algorithm obtained better serendipitous coverage. In the experiments we will describe next (in Section 5.4), the opposite thing happened: Conjugate gradient descent outperformed the reference algorithm by—according to the logged status information—obtaining better serendipitous coverage.

Additional results are shown in the Appendix. The performance gap between the three techniques seems to grow when the target program becomes more complex, but it remains small when either the condition complexity or the nesting complexity is small. This suggests that dynamic test data generation is worthwhile for complex programs, though it may not be worthwhile for programs whose conditions are simple or are not nested.

However, it seems that random test generation levels off after several hundred executions, during which time the GAs show the same level of performance; since evolution has not begun yet, their behavior is essentially random. This is illustrated especially well in Figs. 5 and 14.

5.3.1 The Performance of the Differential GA Compared to the Standard GA

The poor performance of the differential GA (compared to the standard GA) can probably be attributed to its numerical focus, which seems poorly suited to test generation problems. Recall from Section 3.2 that the differential GA makes numerical changes in many variables, while the standard GA, using single-point crossover, leaves most variable values intact during reproduction. The scheme used by the differential GA is desirable in numerical problems, while the standard GA seems more suited to situations where variables represent specific features of a potential solution. The question, therefore, is whether the variables in a test generation problem behave more like the parameters of a numerical function,or if they are more like "features" describing the behavior of the program.

This somewhat cloudy question becomes more concrete if we focus on the part of the program's behavior that is of interest during test generation. The program's input is a set of parameter values, but, in a test generation problem, we generally ignore the output and are only interested in the *control path* taken during execution. In this sense, test generation is not what would normally be thought of as a numerical problem. Indeed, a small change in the input parameters often leads to no change at all in the control path, so the fine parameter adjustments made by the differential GA are often fruitless.

On the other hand, there are many programs where each parameter controls specific aspects of a program's behavior. This may make it more appropriate to think of the parameters as *features*, each one describing a different dimension of the program's behavior. If we regard the parameters as features, we see that the differential GA has a tendency to modify each feature, while the standard GA (with single-point crossover) tends create different combinations of existing features.

This gives the differential GA a disadvantage when it comes to the serendipitous discovery of new inputs. This phenomenon, which we will discuss in the next section, requires that existing control paths (that is, existing features) be preserved, and the differential GA may not be good at doing this.

In our experiments, the disadvantage of the differential GA was sometimes amplified since we insisted on having different optimization methods use roughly the same number of program executions. To make the standard GA execute the program as many times as the differential GA did during its fine-grained and fruitless searches, we increased the standard GA's population size. The standard GA tended to outperform the differential GA even when their populations were the same, and increasing the population size allowed it to *widen* its search while the differential GA remained tightly focused.

To make matters worse for the differential GA, small differences in performance are magnified over time. The failure to find an input that causes one branch of a decision to be executed means that no conditions within that branch can be covered in the future. Thus, the failure to satisfy a single condition can be a greater handicap than it seems.

5.4 b737: Real-World Control Software

In this study, we used the GADGET system on `b737`, a C program which is part of an autopilot system. This code has 69 decision points, 75 conditions, and 2,046 source lines of code (excluding comments). It was generated by a CASE tool.

We generated tests using the standard GA, the differential GA, conjugate gradient descent, the reference gradient descent algorithm, and random test data generation. For each method, 10 attempts were made to achieve complete coverage of the program. For the two genetic algorithms, we made some attempt to tune performance by adjusting the number of individuals in the population and the number of generations that had to elapse without any improvement before the GAs would give up.

For the standard genetic algorithm, we used populations of 100 individuals each and allowed 15 generations to elapse when no improvement in fitness was seen. The probability of mutation was 0.000001. For the differential GA, we used populations of 24 individuals each and allowed 20 generations to elapse when there was no improvement in fitness (the smaller number of individuals allows more generations to be evaluated with a given number of program executions, and we found that the differential GA typically needed more generations because it converges more slowly than the standard GA for these problems). As before, we attempted to generate test cases that satisfy *condition-decision coverage*.

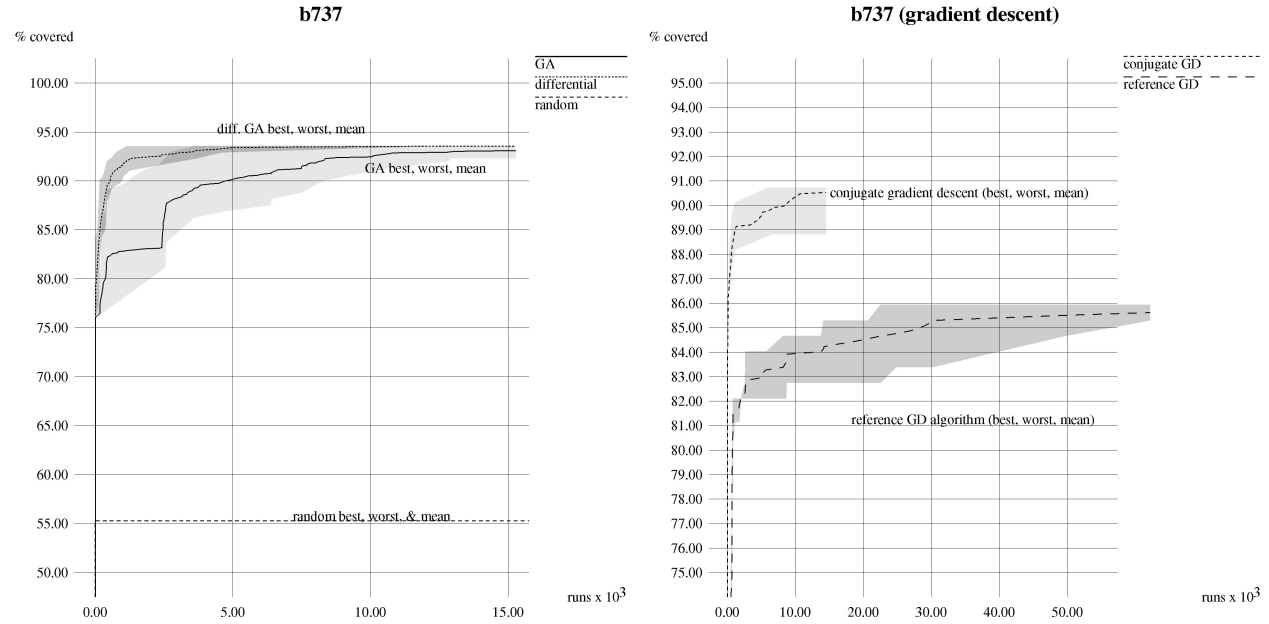


Fig. 8. Coverage plots comparing performance of four systems on the `b737` code. The four curves represent the pointwise mean over 10 attempts by each system to achieve complete coverage. The upper and lower edges of the dark gray region represent the best and worst performances of the standard GA, while the edges of the light gray region are the best and worst performances for the differential GA. The GAs have comparable performance and both show much better performance than random test data generation. The performance of gradient descent was slightly lower; conjugate gradient descent achieved just over 90 percent condition-decision coverage. The vertical axis shows the percentage of the 75 conditions that have been covered. (Note that the two plots do not have the same horizontal or vertical scale.)

First, we tried to achieve condition-decision coverage with the two GAs. Next, we applied random test data generation to the same program. Here, we permitted the same number of program executions as was used by the genetic searches. This amounts to thousands of random tests, one for each time the fitness function was evaluated during genetic search. Note, however, that random test generation stops making progress quickly.

Fig. 8 shows the coverage plot comparing genetic and random test data generation. The graphs show the best, worst, and pointwise mean performance over 10 separate attempts by each system to achieve complete condition-decision coverage.

The best performance is that of the differential GA; eventually, all runs converged to just over 93 percent condition-decision coverage. The best runs of the standard GA also reached this level of performance, but the differential GA was faster. Random test generation only achieved 55 percent coverage.

Though we did not tune the GAs extensively for performance (as we have said), we did try to adjust the population size of both programs in order to obtain comparable resource requirements for both programs. The reason for doing this was to ensure that better or worse performance by one algorithm was not merely the result of a greater or smaller number of program executions. But, during this process, we found that changing the population size only had a small affect on performance. The reason was that the GAs wasted many program executions on fruitless searches and changes in the population size simply changed the resources wasted in this way. Therefore, it is quite possible that the standard GA could have been faster if the population size had been smaller, without a significant decrease in the condition-decision coverage it achieved.

The two gradient descent algorithms were somewhere in the middle, with conjugate gradient descent achieving about 90 percent condition-decision coverage, while the reference algorithm only reached about 85 percent.

5.5 Detailed Analysis of Experiments

For any given execution of the test generator, we can divide the 75 conditions of `b737` into four classes. Some were never covered by the GA, some were covered serendipitously while the GA was trying to cover a different condition, some were covered while the GA was actually working on them, and some were covered by the randomly selected inputs we used to seed the GA initially.

The last class of inputs is reflected in Fig. 8 by the fact that many conditions were already covered almost immediately. For example, the standard GA covered about 76 percent of the conditions right away.

Here, as in our other experiments, most inputs were discovered serendipitously. (Note that, when a test requirement is satisfied serendipitously, it often happens *before* the genetic algorithm makes a concerted attempt to satisfy that requirement. Therefore, the fact that many requirements were satisfied by chance does not imply that the GA would have failed to satisfy them otherwise.)

The fact that most inputs were discovered by luck means that most test requirements *not* satisfied by chance were not satisfied at all. In this respect, the `b737` experiments shed some light on the true behavior of the two different GA implementations. A quick look at parts of the source code elucidates this behavior. We will first explain a typical individual run of the standard GA and then discuss the differential GA.

The execution of the standard GA we examine as a model was selected because its coverage results are close to the mean value of all coverage results produced by the standard GA on b737. In its $11,409$ executions of b737, this run sought to satisfy test requirements on 12 different conditions in the code. Of these 12 attempts, only one was successful. The remaining 11 attempts showed little forward progress during 10 generations of evolution.

While making these attempts, however, the GA coincidentally discovered 14 tests that satisfied requirements other than the ones it was working on at the time. The high degree of coverage that was finally attained was mostly due to those 14 inputs. Indeed, the most successful executions have the shortest runtimes precisely because so many inputs were found serendipitously—many conditions had already been covered by chance before the GA was ready to begin working on them.

Next, we consider a typical execution of the differential GA. During this execution, the $15,982$ executions of b737 involve 25 different attempts to satisfy specific test requirements. The number of attempts depends on which requirements are satisfied and in what order. Therefore, it is not the same for all executions of the test data generator. Again, only one objective is obtained through evolution, though 10 additional input cases, not necessarily prime objectives of the GA, are found.

5.5.1 Where the GAs Failed

It is interesting to consider the conditions that the GAs never successfully covered. The standard GA failed to cover the following eight conditions:

```
for (index = begin;
 index <= end && !termination; index++)
if (((T <= 0.0) || (0.0 == Gain)))
if (o_5)
if (o_5)
if (((o_5 > 0.0) && (o < 0.0)))
if (FLARE)
if (FLARE)
if (DECRB)
```

The differential GA failed to cover these conditions:

```
for (index = begin;
 index <= end && !termination; index++)
if (o)
if (o)
if ((((OP - D2) < 0.0) && ((OP - D1) > 0.0)))
if (o_3) if (((o_5 > 0.0) && (o < 0.0)))
if (FLARE)
if (((!LOCE) || ONCRS))
if (RESET)
```

Most of the decisions not covered only contain a single Boolean variable, signifying a condition that can be either TRUE or FALSE. The technique we use to define our fitness function seems inadequate when the condition contains Boolean variables or enumerated types. For example, if we are trying to exercise the TRUE branch of the condition

```
if (windy) ...
```

we simply make $\Im(x)$ equal to the absolute value of windy. This makes $\Im(x)$ zero when the condition is FALSE and positive otherwise. But, if windy only takes on two values (say 0 and 1), then the fitness function can only have two values as well.

Any two-valued fitness function does not allow the genetic algorithm to distinguish between different inputs that fail to satisfy the test requirement. Genetic search relies on the ability to prefer some inputs over others, so two-valued variables cause problems when they appear within conditions. Our experimental results suggest that this problem is real. With an improved strategy for dealing with such conditionals, GA behavior should improve.

The GAs also failed to cover several conditions not containing Boolean variables, in spite of the fact that such conditions provide the GAs with useful fitness functions. The conditions not covered by the GAs all occurred within decisions containing more than one condition, and this may account for the the GAs difficulties. However, it is also important to bear in mind that these conditions do not tell the whole story since the variables appearing in the condition may be complicated functions of the input parameters.

5.5.2 The Performance of the Random Test Generator

A second question raised by our experiments is why the random test generator performed so poorly. In all of our experiments, the random test generator quickly satisfied a certain percentage of the test requirements and then leveled off, failing to satisfy any further requirements, even though, in some cases, there were thousands of additional program executions.

In many programs, it is not counterintuitive that random test generation performs poorly. The b737 program presents an intuitively striking illustration of this. This program has 186 floating-point input parameters, meaning that there are $2^{11,904}$ possible inputs if a floating-point number is represented with 64 bits. Even a program like triangle, with three floating-point parameters, has 2^{192} possible inputs. With a search space of this size, nothing even approaching an exhaustive search is possible. It is clear that, for any probability density governing the random selection of inputs, one can write conditions that only have a minute probability of being satisfied by those inputs.

Thus, even seemingly straightforward test requirements can be essentially impossible to satisfy using a random test generator. Some parts of the triangle program are only executed when all three parameters are the same, but a random test generator only has one chance in 2^{128} of coming up with such an input if inputs are selected uniformly and independently (e.g., the second and third input parameters have to have the same value as the first, whatever that value happens to be). Since the random test generator creates tests *independently* at random, it is straightforward to determine that the probability of generating an equilateral triangle one or more times during $15,000$ tests is less than 4×10^{-35}. (See [12] for a discussion of the probability of exercising a program feature during testing that was not exercised previously.)

In some cases, the performance of the random test generator might be improved by nonuniform sampling, as discussed in Section 5.2. If the test generator were only allowed to choose integer inputs between 1 and 100—the tester would already have to know that restricting the

inputs to small integers does not preclude finding the desired test—then there would be one chance in $10,000$ of finding a satisfactory input. This would give the random test generator a manageable chance of finding an input whose three parameters are all the same. Still, the same trick would also improve the performance of the GA-driven test generator. The `triangle` example clearly illustrates the advantages of a *directed* search for a satisfactory input.

5.5.3 The Performance of Gradient Descent

The coverage plots for conjugate gradient descent and the reference gradient descent algorithm are shown on the right side in Fig. 8, with a different horizontal and vertical scale than the coverage plot for the GAs. On average, conjugate gradient descent achieved 90.53 percent condition-decision coverage, just slightly less than the genetic algorithms. The reference algorithm achieved 85.51 percent.

In fact, conjugate gradient descent was more expensive than genetic search in these experiments. The reason for this, according to the status information logged during execution, was that conjugate gradient descent had trouble noticing when it was stuck on a plateau. It continued searching there when it should have stopped. This could easily be fixed, but tuning gradient descent for efficiency is our goal in this paper. Needless to say, the reference algorithm was more expensive than any of the others.

The poor performance of the reference algorithm is somewhat surprising. We would expect it to perform at least as well as conjugate gradient descent if the objective function had no local minima or plateaus. The status information logged by the reference algorithm does not show that it encountered local minima, but it did encounter plateaus.

According to the logged status information, this difference in performance was, once again, the result of serendipitous coverage. Below, we will discuss the issue of serendipitous coverage in more detail.

5.5.4 Serendipitous Coverage

Recall that, in the `b737` code, 14 conditions were covered while the GA was trying to satisfy a different condition, while only one condition was covered while the GA was actually trying to cover it. For the differential GA, the ratio was 10 coincidental coverages to one deliberate coverage.

This phenomenon was seen throughout our experiments and it is graphically illustrated in the coverage plots. There are sudden, seemingly discontinuous increases in the percentage of conditions covered. Instantaneous jumps come about because many new conditions are encountered and satisfied immediately when the program takes a new branch in the execution path (this was discussed in Section 5.2). But, sometimes, the sudden increases in coverage are rapid without being instantaneous. In our experiments, this happened when the execution of a new branch provided new opportunities for serendipitous coverage.

The most interesting question raised by this experiment is the following: If the two GAs had so much success with inputs they happened on by chance, then why didn't random test generation perform equally well? We believe that the evolutionary pressures driving the GA to satisfy even one test requirement are strong enough to force the system as a whole to delve deeper into the structure of the code. This means though the GA is not necessarily following the optimal algorithm of grinding through each conditional one after the other to meet its objectives in lock-step manner, it is, in the end, finding good input cases.

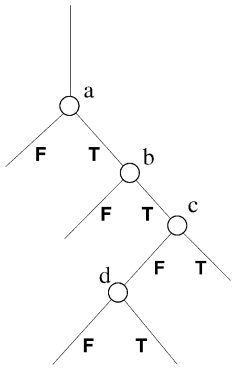


Fig. 9. The flow of control for a hypothetical program. The nodes represent decisions and the goal is to find an input that takes the TRUE branch of decision c.

This argument is illustrated in the diagram in Fig. 9, which represents the flow of control in a hypothetical program. The nodes represent decisions. Suppose that we do not have an input that takes the TRUE branch of the condition labeled c. Because of the coverage-table strategy, GADGET does not attempt to find such an input until decision c can be reached (such an input must take the TRUE branches of conditions a and b). When the GA starts trying to find an input that takes the TRUE branch of c, inputs that reach c are used as seeds. During reproduction, some newly generated inputs will reach c and some will not, but those that do not will have poor fitness values and they will not usually reproduce. Thus, during reproduction, the GA tends to generate inputs that reach c. Until the GA's goal is satisfied, all newly generated inputs that reach c will, by definition, take the FALSE branch and, therefore, they will all reach condition d. Each time a new input is generated that reaches c, there is a possibility it will exercise a new branch of d.

According to this explanation, gradient descent should also benefit from serendipitous coverage and, in fact, we found this to be the case (our experiments with gradient descent were performed after the preceding argument was formulated). In the case of the `b737` experiments, this may also explain why conjugate gradient descent performed better than the reference gradient descent algorithm. Conjugate gradient descent tends to take small steps initially, which means that many of those inputs are likely to reach the same branches as the original seed. The reference algorithm, which takes larger steps, may make more exploratory executions that do not reach the same branches as the seed. (Of course, those inputs will lead to poor objective-function values and that direction of search will not be pursued further.)

In contrast to the inputs generated by genetic search and gradient descent, those generated completely at random may be unlikely to reach d because many will take the FALSE branches of conditions a and b. Therefore, random inputs are less likely to exercise new branches of d.

In the final analysis, both GAs clearly outperform random test data generation for a real program of several thousand lines. This is an encouraging result.

6 Open Research Issues

Our experimental results open a considerable number of research issues. These issues all have at their heart the question: How can our test data generation system be further improved? More specifically, how can we make our system find tests that satisfy an even larger proportion of the test requirements? The handful of issues addressed in this section each have the potential to improve system behavior.

- **Improved handling of binary-valued variables.** The fitness function should deal intelligently with conditions that contain two-valued variables (see Section 5.5.1).
- **Improved handling of inputs that fail to reach the target condition.** When genetic search generates an input that fails to reach the condition that we are currently trying to satisfy, that input is simply given a low fitness value. However, we already have at least one input that reaches the condition because of the way the algorithm is defined. If we assign higher fitnesses to inputs that are closer to reaching the condition, it might be possible to breed more inputs that actually reach it.
- **Special purpose GAs**. Much of the GA literature is concerned with investigating *special purpose* GAs whose parameters and mechanisms are tailored to specific tasks [31]. Results garnered from the differential GA versus standard GA comparison that we made suggest that investigation into *designer* GAs would be profitable. This research would focus on GA failure and investigate ways to avoid running out of steam during test data generation. More work should also be done determining exactly why the standard GA seems to outperform the differential GA.
- **Path selection**. Path selection is the use of heuristics to choose an execution path that simplifies test-data generation (as used by TESTGEN; see Section 2.3). Although path-selection is not vital in our test-generation approach, it may still be the case that some execution paths are better than others for satisfying a particular test requirement. If static or dynamic analysis can provide clues about which paths are best, it will not be difficult to bias a genetic search algorithm toward solutions using those paths. In fact, the work of [16], [4] suggests that such an approach can lead to a noticeable improvement in performance.
- **Higher levels of coverage.** In this paper, we reported on the generation of condition-decision adequate test data. However, higher levels of coverage may further discriminate among different test-generation techniques. It would be interesting to apply our technique to multiple-condition coverage as well as dataflow and mutation-based coverage measures.

6.1 Global Optimization vs. Gradient Descent

In general, gradient descent is faster than global optimization algorithms such as genetic search, so it is preferable when it provides comparable performance. However, the global optimization algorithms almost always provided better performance in our experiments.

Gradient descent can fail in two ways: First, it can encounter local minima and, second, it can encounter plateaus in the objective function. When we analyzed the behavior of the reference algorithm, it appeared that local minima did exist for many of our test adequacy criteria—at least with the way we defined the topology of the problem space—but plateaus appeared to be much more common.

Plateaus can also make a global optimization algorithm fail, and they did so in our experiments. Nonetheless, the global optimization algorithms made up for this failing with serendipitous discoveries of new inputs. In a sense, the extra program executions, which make up the bulk of the extra resources needed for global optimization, were put to good use.

On the basis of our experiments, we are therefore reluctant to conclude that global optimization algorithms performed better *because* of their avoidance of local minima. Rather, it seems that their absence of assumptions about the shape of the objective function, and the extra exploration they did to make up for this lack of assumptions paid off in an unexpected way.

Of course, the setting determines whether or not the performance gain is worth the computational effort. For many programs, it may be important to generate tests quickly, and achieving high levels of coverage may be less important. Indeed, it has been argued that coverage for its own sake is not a worthy goal during software testing. However, high test coverage levels are mandated in some settings. We will also argue, in Section 6.2, that achieving structural test coverage is not the only reason for using automatic test generation. In settings where automated test generation replaces *manual* test generation, the automated approach is clearly desirable, and we hope that this paper will constitute progress toward making it feasible in settings where programs are complex and manual test generation is especially laborious.

6.2 Further Applications of Test-Data Generators

In the long term, there are a number of interesting potential applications of test data generation that are not related to the satisfaction of test adequacy criteria. Often, we would like to know whether a program is capable of performing a certain action, whether or not it was meant to do so. For example, in a safety-critical system, we want to know whether the system can enter an unsafe state. When security is a concern, we would like to know if the program can be made to perform one or more undesirable actions that constitute security breaches. Even in standard software testing, one could conceivably perform a search for inputs that cause a program to fail, instead of simply trying to exercise all features of the program. References [22], [23] and [24] discuss a number of such extensions of dynamic test generation and give more detail than we do here about their implementation.

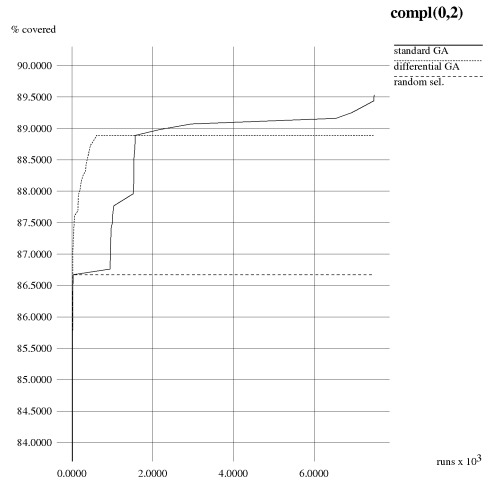


Fig. 10. $compl(0,2);100$ percent coverage represents coverage of 60 conditions.

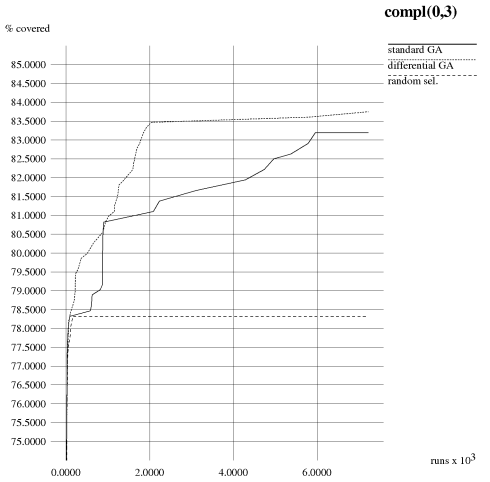


Fig. 11. $compl(0,3);100$ percent coverage represents coverage of 45 conditions.

The genetic search techniques we are developing can be applied in all of these areas, although we expect that each area will present its own challenges and pitfalls. To our knowledge, the only test-data generation systems that can be used on real programs are our own and that of [5]; since both are recent developments, it has not been possible to explore many of the less obvious applications of test data generators. Thus, the ability to automatically satisfy test criteria will open an enormous number of new avenues for investigation.

6.3 Threats to Validity

For the most part, this paper simply reports what we observed in our experiments. By performing each experiment several times with different random number seeds, we tried to ensure that the observations we reported were not extraordinary phenomena, and we can say with a certain confidence that similar results would be obtained if the same experiments were run with different initial seeds.

However, we do not know of any legitimate basis for generalizing about one program after observing the behavior of another nor for selecting programs at random in a way that permits statistical conclusions to be drawn about a broader class of programs. Therefore, we have not tried to obtain, say, a statistical sample of control programs to use in Section 5.4 nor have we tried to generalize about programs with different nesting complexities or condition complexities by generating many random programs with the same $compl(\cdot,\cdot)$ parameters in Section 5.3. It follows, however, that our results in Section 5.4 do not predict the performance of genetic search in automatic test generation for all control software nor do the results in Section 5.3 predict its performance for all software with a given nesting complexity and condition complexity.

Our measurements were made in two ways. First, we kept a log of certain information while test generation was in progress. Our logs show the value of the objective function when the program under test is executed, they record when new coverage criteria are satisfied, and, in some cases, they record other information as well, such as the fact that the reference gradient descent algorithm is reducing its step size. Coverage results are obtained using the commercial coverage tool DeepCover and they are recorded each time a new criterion is satisfied; this allows us to construct convergence plots.

Although most of what we reported is simply what we observed, some conclusions about the shape of the objective function was obtained indirectly via the log files. Thus, when we report that a test generation hits a plateau in the objective function, we mean that the algorithm was unable to find any inputs that caused the objective function value to change and not that there were no such inputs. Likewise, when we say that the objective function was not quadratic, we mean that a discrete set of points—those used as inputs by the test-generator—did not fit a quadratic curve well.

7 Conclusions

In this paper, we have reported on results from four sets of experiments using dynamic test data generation. Test data were generated for programs of various sizes, including some that were large compared to those usually subjected to test data generation. To our knowledge, we present results for the largest program yet reported in the test generation literature. The following are some salient observations of our study:

- In our experiments, the performance of random test generation deteriorates for larger programs. In fact, it deteriorates faster than can be accounted for simply by the increased number of conditions that must be covered. This suggests that satisfying individual test requirement is harder in large programs than in small ones.

  Moreover, it implies that, as program complexity increases, nonrandom test generation techniques become increasingly desirable, in spite of the greater simplicity of implementing a random test generator.

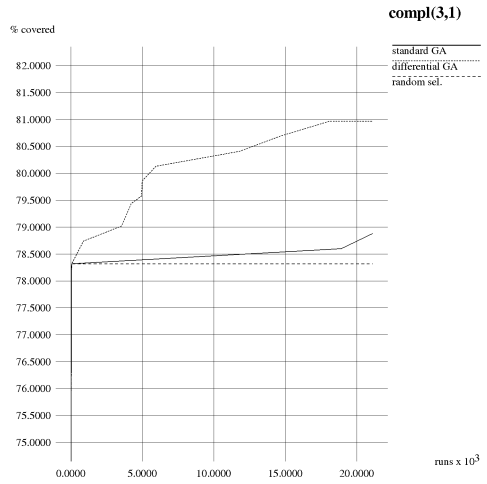


Fig. 12. $compl(3,1)$; the differential GA outperforms the other two techniques, which occured rarely in our experiments. 100 percent coverage represents coverage of 60 conditions.

- Although the standard genetic algorithm performed best overall, there were programs for which the differential GA performed better. For most of the programs, a fairly high degree of coverage was achieved by at least one of the techniques. From the standpoint of combinatorial optimization, it is hardly surprising that no single technique excels for all problems, but, from the standpoint of test-data generation, it suggests that comparatively few test requirements are intrinsically hard to satisfy, at least when condition-decision coverage is the goal. Apparently, a requirement that is difficult to cover with one technique may often be easier with another.
- Serendipitous satisfaction of new test requirements can play an important role. In general, we found that the most successful attempts to generate test data did so by satisfying many requirements coincidentally. This coincidental discovery of solutions is facilitated by the fact that a test generator must solve a number of similar problems, and it may lead to considerable differences between dynamic test data generation and other optimization problems.

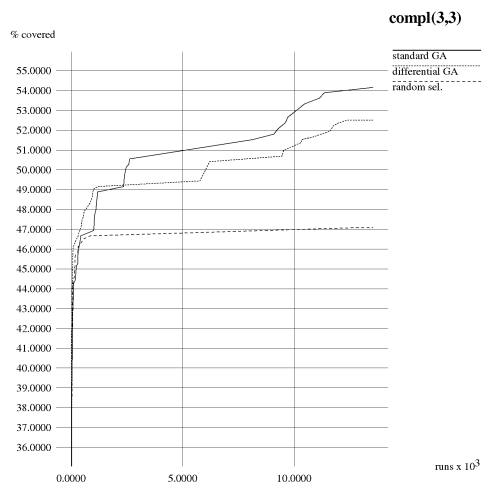


Fig. 13. $compl(3,3);100$ percent coverage represents coverage of 45 conditions.

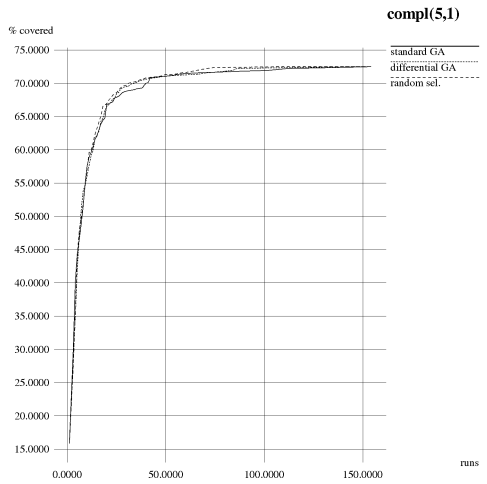


Fig. 14. $compl(5,1);100$ percent coverage represents coverage of 60 conditions. All the requirements that are ever satisfied are satisfied in the early stages of the test generation process. The behavior of the GAs is essentially random because no evolution has taken place yet.

APPENDIX

RESULTS FOR SYNTHETIC PROGRAMS

This appendix shows the results of further experiments described in Section 5.3. Several interesting features, such as the remarkably poor performance of the standard GA for the program with $compl(3,1)$ and the similar performance of all three methods for $compl(5,0)$, also appeared in several repetitions of the experiment using different GA parameters and different seeds for random number generation. This suggests (not surprisingly) that the *compl* metric does not capture everything needed to predict the performance of the test generators.

As in Section 5.3, the differential GA used 30 individuals and gave up on satisfying a given test requirement if 10 generations elapsed with no progress. The standard GA was also instructed to give up if no progress was made in 10 generations, and the population size was adjusted to give the same number of target-program executions as the

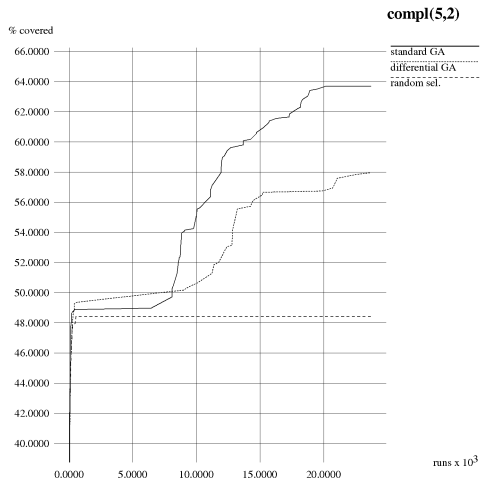


Fig. 15. $compl(5,2)$; there are a total of 60 conditions to cover.

TABLE 6
Performance of Conjugate Gradient Descent and the Reference Gradient Descent Algorithm on the Off-Diagonal Synthetic Programs

	compl(0,2)	*compl(0,3)*	*compl(3,1)*	*compl(3,3)*	*compl(5,1)*	*compl(5,2)*
CGD	85.83	80.0	78.74	41.25	72.5	53.79
ref.	89.25	84.03	81.67	55.28	72.5	62.96

Overall, the techniques are comparable to one another and their performance is somewhat below that of genetic search A remarkable exception is $compl(3,3)$, *where the reference algorithm outperformed all others. The reference algorithm also outperformed conjugate gradient descent on* $compl(0,3)$, *which may be because of the interpolation technique we used for conjugate gradient descent.*

differential GA. This resulted in the following population sizes for the standard GA (shown in Figs. 10, 11, 12, 13, 14, and 15, respectivly):

$$compl(0,2):270, compl(0,3):160, compl(3,1):320,$$
$$compl(3,3):280, compl(5,1):340, \text{ and } compl(5,2):240.$$

The performance for each size is reported in Table 6.

ACKNOWLEDGMENTS

The authors would like to thank Berent Eskikaya, Curtis Walton, Greg Kapfhammer, and Deborah Duong for many helpful contributions to this paper. Tom O'Connor and Brian Sohr contributed the GADGET acronym. Bogdan Korel provided the programs analyzed in Section 5.2. This research has been made possible by the US National Science Foundation under award number DMI-9661393 and the US Defense Advanced Research Projects Agency (DARPA) contract N66001-00-C-8056.

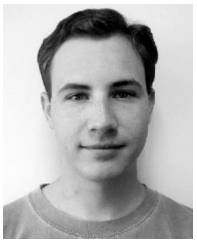
REFERENCES

[1] W. Miller and D.L. Spooner, "Automatic Generation of Floating Point Test Data," *IEEE Trans. Software Eng.,* vol. 2, no. 3, pp. 223–226, Sept. 1976.
[2] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. Software Eng.,* vol. 16, no. 8, pp. 870–879, Aug. 1990.
[3] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini, "Choosing a Testing Method to Deliver Reliability," *Proc. 19th Int'l Conf. Software Eng. (ICSE '97),* pp. 68–78, May 1997.
[4] R. Ferguson and B. Korel, "The Chaning Approach for Software Test Data Generation," *ACM Trans. Software Eng. Methodology,* vol. 5, no. 1, pp. 63–86, Jan. 1996.
[5] M.J. Gallagher and V.L. Narasimhan, "Adtest: A Test Data Generation Suite for Ada Software Systems," *IEEE Trans. Software Eng.,* vol. 23, no. 8, pp. 473–484, Aug. 1997.
[6] J.H. Holland, *Adaption in Natural and Artificial Systems.* Ann Arbor, Mich.: Univ. of Michigan Press, 1975.
[7] S. Kirkpatrick, C.D. Gellat Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4,598, pp. 671–680, May 1983.
[8] F. Glover, "Tabu Search Part I, II," *ORSA J. Computing,* vol. 1, no. 3, pp. 190–206, 1989.
[9] J. Horgan, S. London, and M. Lyu, "Achieving Software Quality with Testing Coverage Measures," *Computer,* vol. 27, no. 9, pp. 60–69, Sept. 1994.
[10] J. Chilenski and S. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Eng. J.,* pp. 193–200, Sept. 1994.
[11] R. DeMillo and A. Mathur, "On the Uses of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software," Technical Report SERC-TR-92-P, Purdue Univ., 1992.
[12] R. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.,* vol. 16, no. 12, pp. 1402–1411, Dec. 1990.
[13] E.J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Trans. Software Eng.,* vol. 17, no. 7, pp. 703–711, July 1991.
[14] E.J. Weyuker, "Axiomatizing Software Test Adequacy," *IEEE Trans. Software Eng.,* vol. 12, no. 12, pp. 1128–1137, Dec. 1986.
[15] P.G. Frankl and E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods," *IEEE Trans. Software Eng.,* vol. 19, no. 3, pp. 202–213, Mar. 1993.
[16] B. Korel, "Automated Test Data Generation for Programs with Procedures," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 209–215, 1996.
[17] L.A. Clarke, "A System to Generate Test Data Symbolically and Execute Programs," *IEEE Trans. Software Eng.,* vol. 2, no. 3, pp. 215–222, Sept. 1976.
[18] C.V. Ramamoorty, S.F. Ho, and W.T. Chen, "On the Automated Generation of Program Test Data," *IEEE Trans. Software Eng.,* vol. 2, no. 4, pp. 293–300, Dec. 1976.
[19] J. Offutt, "An Integrated Automatic Test Data Generation System," *J. Systems Integration,* vol. 1, pp. 391–409, 1991.
[20] W.H. Deason, D.B. Brown, K-H. Chang, and J.H. Cross II, "A Rule-Based Software Test Data Generator," *IEEE Trans. Knowledge and Data Eng.,* vol. 3, no. 1, pp. 108–117, Mar. 1991.
[21] K.H. Chang, J.H. Cross II, W.H. Carlisle, and S.-S. Liao, "A Performance Evaluation of Heuristics-Based Test Case Generation Methods for Software Branch Coverage," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 6, no. 4, pp. 585–608, 1966.
[22] N. Tracey, J. Clark, and K. Mander, "The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach," *Proc. Int'l Workshop Dependable Computing and Its Applications (DCIA),* pp. 169–180, Jan. 1998.
[23] N. Tracey, J. Clark, and K. Mander, "Automated Program Flaw Finding Using Simulated Annealing," *Proc. Int'l Symp. Software Testing and Analysis, Software Eng. Notes,* pp. 73–81, Mar. 1998.
[24] H. Tracey, J. Clark, K. Mander, and J. McDermid, "An Automated Framework for Structural Test-Data Generation," *Proc. Automated Software Eng. '98,* pp. 285–288, 1998.
[25] C.C. Michael, G.E. McGraw, and M.A. Schatz, "Genetic Algorithms for Dynamic Test Data Generation," *Proc. Automated Software Eng. '97,* pp. 307–308, 1997.
[26] C.C. Michael, G.E. McGraw, and M.A. Schatz, "Opportunism and Diversity in Automated Software Test Data Generation," *Proc. Automated Software Eng. '98,* pp. 136–146, 1998.
[27] A.C. Schultz, J.C. Grefenstette, and K.A. DeJong, "Test and Evaluation by Genetic Algorithms," *IEEE Expert,* pp. 9–14, Oct. 1993.
[28] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C.* New York: Cambridge Univ. Press, 1991.
[29] J. Skorin-Kapov, "Tabu Search Applied to the Quadratic Assignment Problem," *ORSA J. Computing,* vol. 2, no. 1, pp. 33–41, Winter 1990.
[30] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, Mass.: Addison-Wesley, 1989.
[31] M. Mitchell, *An Introduction to Genetic Algorithms.* Cambridge, Mass.: MIT Press, 1996.
[32] *Foundations of Genetic Algorithms.* G. Rawlins, ed. San Mateo, Calif.: Morgan Kaufmann, 1991.
[33] R. Storn, "On the Usage of Differential Evolution for Function Optimization," *Proc. North Am. Fuzzy Information Prossessing Soc., (NAFIPS '96),* pp. 519–523, June 1996.
[34] S.K. Park and K.W. Miller, "Random Number Generators: Good Ones are Hard to Find," *Comm. ACM,* vol. 31, no. 10, pp. 1192–1201, Oct. 1988.

[35] R.A. DeMillo and A.J. Offutt, "Experimental Results from an Automatic Test Case Generator," *ACM Trans. Software Eng. Methodology*, vol. 2, no. 1, pp. 215–222, Jan. 1993.
[36] J.D. Musa, "Operational Profiles in Software Engineering," *IEEE Software*, vol. 10, no. 2, pp. 14–332, 1993.

Christoph C. Michael received the BA degree in physics from Carleton College, Minnesota, in 1984, and the MSc and PhD degrees in computer science from the College of William and Mary in Virginia, 1993. He is a senior research scientist at Cigital. He has served as prinicpal investigator on software assurance grants from the US National Institute of Standards and Technology's Advanced Technology Program and the US Army Research Labs, as well as software security grants from the US Defense Advanced Research Projects Agency. His current research includes information system intrusion detection, software test data generation, and dynamic software behavior modeling. He is a member of the IEEE.

Gary McGraw is the vice president of corporate technology at Cigital (formerly, Reliable Software Technologies) where he pursues research in software security while leading the Software Security Group. He has served as principal investigator on grants from the US Air Force Research Labs, the Defense Advanced Research Projects Agency, the US National Science Foundation, and the US National Institute of Standards and Technology's Advanced Technology Program. He also chairs the National Information Security Research Council's Malicious Code Information Security Science and Technology Study Group. He coauthored *Java Security* (Wiley, 1996), *Software Fault Injection* (Wiley, 1997), and *Securing Java* (Wiley, 1999), and is currently writing a book entitled *Building Secure Software* (Addison-Wesley, 2001). He is a member of the IEEE.



Michael A. Schatz graduated summa cum laude with a BS degree in mathematics from Case Western Reserve University in 1996 and an MS degree in computer engineering from Case Western Reserve University in 1997. He is a senior research associate at Cigital (formerly known as Reliable Software Technologies). He has worked on numerous projects in both research and development roles. These projects include experimentation with using fault injection to find security vulnerabilities, using genetic algorithms to generate test data for programs, and augmenting the capabilities of Reliable Software Technologies's coverage tool. He has coauthored articles for *Dr. Dobbs* and a number of research papers.