Automated Software Test Data Generation for Complex Programs

Christoph Michael & Gary McGraw[1]
Reliable Software Technologies
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
`http://www.rstcorp.com`
`{ccmich,gem}@rstcorp.com`

Abstract

We report on GADGET, a new software test generation system that uses combinatorial optimization to obtain condition/decision coverage of C/C++ programs. The GADGET system is fully automatic and supports all C/C++ language constructs. This allows us to generate tests for programs more complex than those previously reported in the literature. We address a number of issues that are encountered when automatically generating tests for complex software systems. These issues have not been discussed in earlier work on test-data generation, which concentrates on small programs (most often single functions) written in restricted programming languages.

1 Dynamic test data generation

In this paper, we introduce the GADGET system, which uses a test data generation paradigm commonly known as *dynamic test data generation.* Dynamic test data generation was originally proposed by [Miller and Spooner, 1976] and then investigated further with the TESTGEN system of [Korel, 1990, Korel, 1996], the QUEST/Ada system of [Chang et al., 1996], and the ADTEST system of [Gallagher and Narasimhan, 1997]. This paradigm treats parts of a program as functions that can be evaluated by executing the program, and whose value is minimal for those inputs that satisfy test adequacy criterion such as code coverage. In this way, the problem of generating test data reduces to the better-understood problem of function minimization.

Standard approaches to dynamic test data generation suffer from two main problems:

1. *Research prototypes place severe constraints on the language being analyzed (*e.g.*, disallowing function calls or procedures) so that programs can more easily be instrumented by hand.* QUEST/Ada requires the program under test to be instrumented by hand. TESTGEN only allows programs written in a subset of the PASCAL language. The problem with such limitations is that they prevent one from studying complex programs. The unchallenging demands of simple programs can make naïve schemes like random test generation appear to work better than they actually do [McGraw et al., 1997].

2. *The function minimization techniques applied are often overly simplistic.* ADTEST and TESTGEN use gradient descent to perform function minimization, and this technique suffers when the objective function contains local minima. Although quantitative results were not reported on the performance of ADTEST, [Gallagher and Narasimhan, 1997] states that local minima cause difficulties for that system. TESTGEN's gradient descent system performed well in [Korel, 1996], but it was aided by a heuristic path selection strategy.

As a result of these two weaknesses, programs for which test data are generated in the literature are traditionally small, with few conditionals, little nesting, and simple control flow structures.

Function minimization using gradient descent suffers from some well understood weaknesses. In our work we apply more sophisticated techniques for function minimization—genetic search [Holland, 1975] and simulated annealing [Kirkpatrick et al., 1983]. In this paper, we compare the performances of simulated annealing, two implementations of genetic algorithms, and a form of gradient descent when they are applied to dynamic test data generation. We use a random test data generator to create a baseline for our comparisons.

By automating instrumentation of a program for analysis, we are able to analyze programs including all C/C++ language constructs, including function and method calls. This opens the door for experimentation with more complex programs than those previously studied.

[1]McGraw is correspondence author.

Our experiments demonstrate a potentially important difference between dynamic test data generation and other function minimization problems. On the programs we tested, *coincidental* discovery of test inputs satisfying new criteria was more common than their deliberate discovery. Although one might expect *random* test generation to be good at discovering things by coincidence, especially given a set of standard constraints [Duran and Natfos, 1984], it did not perform well in our experiments. The guided search performed by more sophisticated techniques is good at setting up the coincidental discovery of tests satisfying new criteria, since these methods tend to concentrate on restricted areas of the input space.

1.1 Code coverage as test data generation criteria

Empirical results indicate that tests selected on the basis of test adequacy criteria such as code coverage are good at uncovering faults [Horgan et al., 1994, Chilenski and Miller, 1994]. Furthermore, test adequacy criteria are objective measures by which the quality of software tests can be judged. Neither of these benefits can be realized unless test data that satisfy the adequacy criteria can be found. Therefore there is a need to generate such tests automatically.

In practice, most 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 criteria are usually called *coverage analyses*, because certain features of the source code are to be covered by the tests. The example given above describes *statement coverage*.

There is a hierarchy of increasingly complex coverage criteria having to do with the conditional statements in a program. 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. 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.

1.2 Test generation as function minimization

Our approach, after Korel, is based on the idea that parts of a program can be treated as functions. One can execute the program until a certain location in the code is reached, record the values of one or more variables at that location, and treat those values as though they were the value of a function. 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 on the input x. Then the function:

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

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 the objective function $\mathfrak{F}(x)$.

This is unfortunately an oversimplification, because line 324 may not be reached for some inputs. There are two common solutions to this problem. First, one can treat the problem of reaching the desired location as a *sub*problem that must be solved before the minimization of $\mathfrak{F}(x)$ can commence [Korel, 1990]. Second, one can amend the definition of $\mathfrak{F}(x)$ so that it will have a very large value whenever the desired condition is not reached [Gallagher and Narasimhan, 1997]. Another possibility (the one that we adopt) is to implement an opportunistic strategy that seeks to cover whatever conditions it can reach [Michael et al., 1997].

Korel's subgoal chaining 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. 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 [Gallagher and Narasimhan, 1997], 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. The ADTEST system begins by trying to satisfy the first condition on the path, and the second condition is added only after the first condition can be satisfied. As more conditions are reached, they are incorporated in the function that the algorithm seeks to minimize.

1.2.1 Coverage tables and opportunism

The QUEST/Ada system of [Chang et al., 1996] creates test data using rule-based heuristics. For example, one rule causes values of parameters 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 each branch and marks those that have been successfully covered. The table is consulted during analysis to determine which branches to target for testing. Partially-covered branches are always chosen over completely–non-covered branches. We independently developed the coverage-table strategy, and use it in GADGET.

This procedure can be illustrated using the following code fragment, which comes from a control system:

```
1: if (state_error > state_bndry[0])
2:       rule_area = area (1.0, (state_bndry[0] - state_bndry[1]));
3:  else if ((state_error > state_bndry[2]))
4:       state_weight = rule (state_error, state_bndry[2], state_bndry[0]);
5:  else ...
```

In order to reach the `if` condition on line 3, a test case must cause the condition on line 1 to be FALSE. Any conditions on line 5 can only be reached when the condition on line 1 and the condition on line 3 are both FALSE.

Decision	TRUE	FALSE
1	X	X
3	-	X

Table 1: A sample coverage table after Chang. This table can be automatically generated from the code fragment shown above. Such tables are used to decide where to apply function minimization during automatic test data generation.

If the goal of test generation is to exercise all branches in the code, then there must both be test cases that cause the first condition to be TRUE as well as some that cause it to be FALSE. Therefore, when we have achieved branch coverage of line 1, we will already have at least one test case that reaches the condition on line 3. We are thus ready to begin looking for test cases that cover both branches of the decision on line 3. The situation is illustrated by Table 1.

Coverage tables provide a strategy for dealing with the situation where a desired condition is not reached. Instead of picking a particular 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.

2 GADGET

GADGET's name reflects the fact that it originally used only genetic algorithms to perform function minimization (GADGET is an acronym expanding to "Genetic Algorithm Data GEneration Tool.") GADGET currently implements four optimization techniques for test generation: simulated annealing, gradient descent, a standard genetic algorithm, and a differential genetic algorithm. GADGET can also generate tests at random (providing our baseline approach).

GADGET automatically generates test data for arbitrary C/C++ programs, with no limitations on the permissible language constructs and no requirement for hand-instrumentation. We report test results for complex programs containing up to 2000 lines of source code with complex, nested conditionals. To our knowledge, these are the most complex programs for which results have been reported. By contrast standard programs reported in the literature average 30 lines of code and have relatively simple conditionals [Yin et al., 1997,Korel, 1996,Chang et al., 1996]. Results of GADGET runs on small programs can be found in [McGraw et al., 1997]. We take advantage of GADGET's capability for processing complex programs by examining the impact of program complexity on the problem of dynamic test data generation.

Our **gradient descent** 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 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 ten parameters (creating a ten-dimensional input space) has twenty 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 depended on the experiment, and whose standard deviation was half of the mean.

Our **simulated annealing** algorithm is close to the standard Metropolis algorithm described in [Kirkpatrick et al., 1983]. For a given seed input, a randomly selected neighbor is generated with one of the same neighborhood functions used by gradient descent (the step size is also randomly selected, as above). If the new input results in an improved objective function, it becomes the new seed. If it does not improve the objective function, it still becomes the new seed with probability $e^{-d/T}$, where d measures the worsening of the objective function value, and T is a parameter usually called the temperature of the system. T gradually decreases with time. Moves that make the objective function worse are more likely when the temperature is higher, and they are impossible when the temperature reaches zero. At zero temperature, the Metropolis algorithm allows a move to a new seed that does not change the objective function value. Because we encountered many flat regions in the objective function, we disallowed such moves at zero temperature unless none of the inputs in the neighborhood led to an improvement.

The first of our **genetic algorithms** (GA) is standard. It begins by encoding each input as a string of bits. A population of such inputs is randomly generated. In the first step, *evaluation*, the objective function is evaluated for each input, giving the input a fitness value. The next step, *selection*, is used to find two inputs that will be mated to contribute to the next generation. The two inputs are selected at random, but each input's probability of being chosen is proportional to its fitness. The third step is *crossover* (or recombination). If an input contains n bits, the crossover point is a randomly selected integer c, $0 < c < n$. Two offspring A and B are created in such a way that the first c bits of A are copied from the first parent, while the first c bits of B are copied from the second. After the crossover point, each child takes its bits from the remaining parent, so that A's last $n - c$ bits come from the second parent, while the last $n - c$ bits of B come from the first. In addition to evaluation, selection, and recombination, genetic algorithms use *mutation* to guard against the permanent loss of information. Mutation simply results in the infrequent flipping of a bit within an input.

Our second genetic algorithm is the **differential GA** described in [Storn, 1996]. 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.

3 Experimental results

In this section, we describe several experimental results obtained with GADGET. All experiments report on condition/decision coverage criteria. We begin with a simple program, `triangle`, for which results are often reported in the literature. The second set of results was obtained from a suite of synthetic programs with varying complexity. These experiments were designed to explore the effect of program complexity on the difficulty of automatic test data generation. Our final set of results comes from a complex component of a Boeing 737 autopilot control program with about 2,000 lines of code. `B737` includes both complex control structures and deeply nested conditionals.

3.1 Simple programs, including `triangle`

We begin our GADGET experimentation on a set of simple functions much like those reported in the literature [Chang et al., 1996, Korel, 1990, Yin et al., 1997]. These programs are roughly of the same complexity order, averaging 30 lines of code and all having relatively simple decisions. Complete results for: `Binary search`, `Bubble sort`, `Date range`, `Euclidean greatest common denominator`, `Insertion sort`, `Computing the median`, `Quadratic formula`, and `Warshall's algorithm` can be found in [McGraw et al., 1997]. Averages from the results reported there show that random achieves 88.28% coverage, the standard GA 93.159%, and the differential GA 95.93%. Since the addition of simulated annealing and gradient descent to GADGET we have not experimented with simple programs.

Figure 1 shows how the three of our five algorithms—standard GA, differential GA, and random—perform on the `triangle` program. The GA exhibits the best performance, something that holds across the board in simple program experiments. In general the differential GA shows slower convergence to a solution. It is taking advantage of a longer focus on a local area of the search space.

Our results for random test case generation resemble those reported elsewhere for cases where the code being analyzed was relatively simple. In [Chang et al., 1996], random test data generation was reported to cover 93.4% of the conditions on average. Although its worst performance—on a program containing 11 decision points—was 45.5%, it outperformed most of the other test generation schemes that were tried. Only symbolic execution had better performance for these programs. [Korel, 1996] reports on eleven programs averaging just over 100 lines of code. Overall, random test data generation was fairly successful, achieving 100% statement coverage for five programs, and averaging 76% coverage on the other six.

It is also interesting to compare our results with those obtained by [Korel, 1996] for three slightly larger programs. Simple branch coverage was the goal. Random test generation achieved 67%, 68%, and 79% coverage, respectively, on the three programs analyzed. Symbolic test generation achieved 63%, 60%, and 90% coverage, while dynamic test generation achieved 100%, 99%, and 100% coverage.

These results show a common trend: random test generation has at least an adequate performance on such programs, but for larger programs or more demanding coverage criteria (including condition/decision coverage), its performance deteriorates. The programs used in [Korel, 1996] were larger than those used in [Chang et al., 1996], and random test generation had poorer performance on the larger programs.

The results reported in [McGraw et al., 1997] use small programs, though the test-adequacy criterion is more stringent. In some cases (like the `triangle` program) random test generation is far less adequate even though resource limitations are not a telling factor: the final 90% of the randomly-generated tests failed to satisfy any new coverage criteria. In our subsequent experiments on larger programs, we find this to be a continuing trend: program size, program complexity, and coverage level all decrease the percentage of adequacy criteria that can be satisfied using random test generation.

Since the percentage of adequacy criteria satisfied by random test generation actually goes down when the programs become more complex, the suggestion is 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 confirmed in our later experiments.

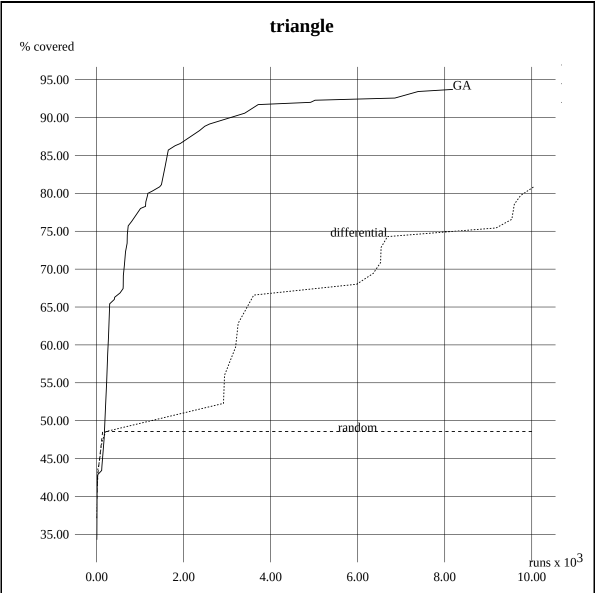

Figure 1: A convergence graph for the `triangle` program. The graph shows how performance (in terms of percent CDC coverage) improves as the number of runs increases. The curves represent the pointwise mean performance over five runs for each system.

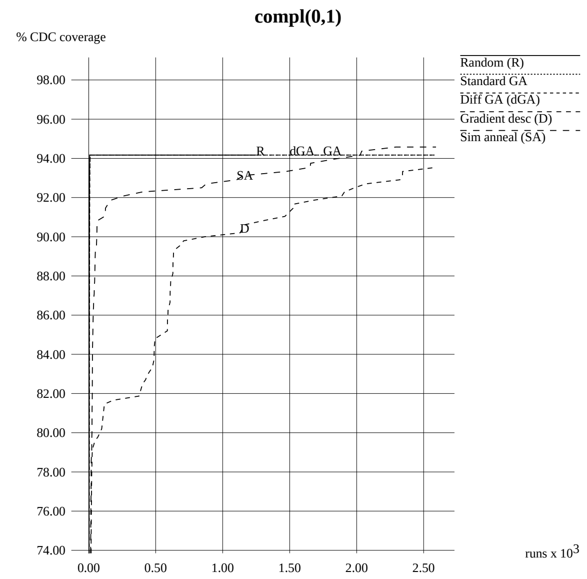

Figure 2: Random test generation and the GAs perform well on the simple program, *compl(0,1)*. Simulated annealing and gradient descent need many more executions to attain the same performance. Curves in Figures 1–3 represent the pointwise mean over four runs for each system.

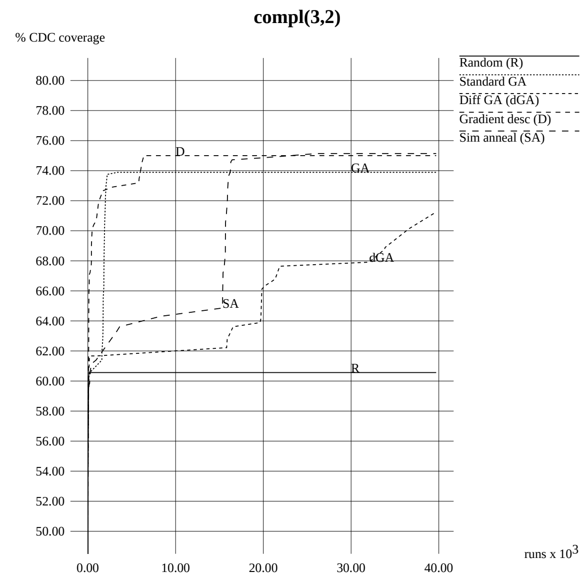

Figure 3: As complexity increases, simulated annealing and gradient descent begin to out-pace the standard GA, but they are still somewhat more expensive. The differential GA's performance lagged. *compl(3,2)*.

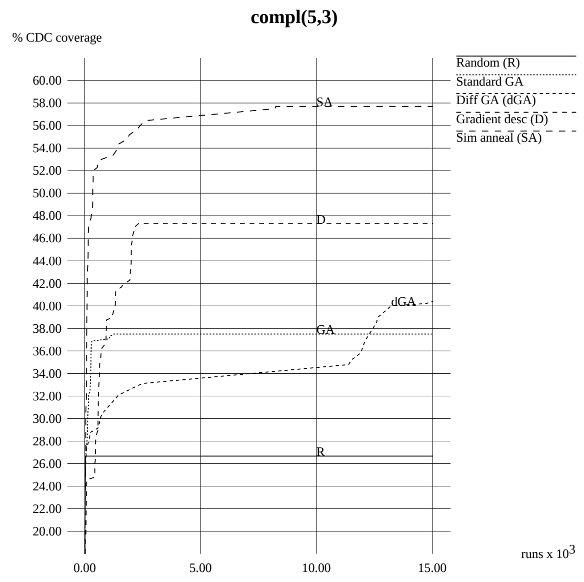

Figure 4: High complexity programs are hard for all generators, but simulated annealing has the best performance. Gradient descent outperforms the GAs but is slightly more expensive than the standard GA. *compl(5,3)*.

3.2 The role of complexity

In our second set of experiments, we create synthetic programs of varying complexity and use GADGET to generate test data satisfying condition/decision coverage. We are particularly interested in controlling two characteristics of programs: 1) how deeply conditional clauses were nested (we call this the *nesting factor*), and 2) the number of Boolean conditions in each decision (which we call *condition factor*). The second parameter controls the complexity of individual conditional expressions; for example the decision `((i <= 0) || (j <= 0) || (k <= 0))` contains three conditions, and thus has a condition factor of three. We can classify programs according to their complexity with a function *compl(nesting factor, condition factor)*. In our experiments, programs were generated with complexities $compl(0,1)$, $compl(3,2)$, and $compl(5,3)$. All of these programs have ten input parameters, so they lead to a ten-dimensional search space for the optimization algorithms.

All of our optimization algorithms use the same parameters for each of the three synthetic programs. The two GA's have a population size of thirty, and they abandon a search if no progress is made in ten generations. Gradient descent and simulated annealing use step sizes selected according to a Gaussian distribution with mean 50 and standard deviation 25. For simulated annealing, the initial temperature is 0.1 and it is decreased in steps of 0.01. The temperature is decreased when twenty moves produce no improvement in the objective function.

Figures 2–4 show the pointwise mean performance over four runs of each test generation technique. For the simplest program, random test generation and the GAs quickly achieve high coverage (they are not distinguishable in the plot). Although gradient descent catches up, and simulated annealing ultimately surpasses the other techniques, both take many executions to do so. When we examined the details of each execution, we find that the GAs perform well because they often manage to satisfy new criteria coincidentally — that is, they often find inputs satisfying one criterion while they are searching for an input that satisfies another.

Gradient descent and simulated annealing are not as good at uncovering useful inputs coincidentally. They work as hard on the simple program as they do on the complex ones, but the extra effort is not needed in this case. Ultimately, they do as well as random test generation and the GAs, but they are unnecessarily expensive.

Figure 3 shows results for a program of intermediate complexity, $compl(3,2)$. Random test generation does not do as well as before. The standard GA continues to discover many inputs by coincidence, but it quickly levels off. The differential GA performs a more focused search than the standard GA, and it fails to find as many inputs by coincidence. Gradient descent and simulated annealing perform better than the GAs and use fewer resources than the differential GA, but they are still more expensive than the standard GA.

Finally, Figure 4 shows the results for a program with $compl(5,3)$. Here, simulated annealing is clearly more successful than any of the other techniques. Gradient descent is almost as successful, though the standard GA outperforms it in the early stages. The differential GA ultimately outperforms the standard GA.

The total number of executions was smaller in the $compl(5,3)$ experiment than in $compl(3,2)$. The reason for this is that none of the test generators attempt to satisfy conditions they cannot reach. In the $compl(3,2)$ experiment, more conditions are reached, so all the generators do more work.

We cannot conclude from these experiments that GAs perform poorly on complex programs compared to simulated annealing. With each of the optimization techniques we used, it is standard for practitioners to tune parameters like the population size, annealing schedule, and so on, in order to get better performance. Mere coincidence may account for the fact that our simulated annealing technique is better tuned for the synthetic complex programs than the GAs are. Although we made some attempt to tune the optimizers, nothing is known about how they *should* be tuned for more complex programs. This is, after all, the first time most of these techniques have been used for test data generation.

We *can* conclude, however, that program complexity has a drastic affect not only on the performance of our various optimization techniques, but also on which strategies are best. Clearly, more research is needed on the relationships between program complexity, search space characteristics, and dynamic test generation strategies.

We can also conclude that coincidental satisfaction of new criteria plays an important role, especially in simpler programs. This allows the genetic algorithms to perform as well as random test generation on simple programs, even though one might otherwise expect them to do as much unnecessary work as gradient descent or simulated annealing.

3.3 b737

In our final study, we use the GADGET system on `b737`, a C program which is part of an autopilot system. This code has 69 decision points and 2046 source lines of code (excluding comments), and it was generated by a CASE tool. The program has a 186-dimensional input space. Figure 5 shows the function call graph of `b737` which imparts some idea of its complexity as a program. (A similar graph for `triangle` has only four nodes.)

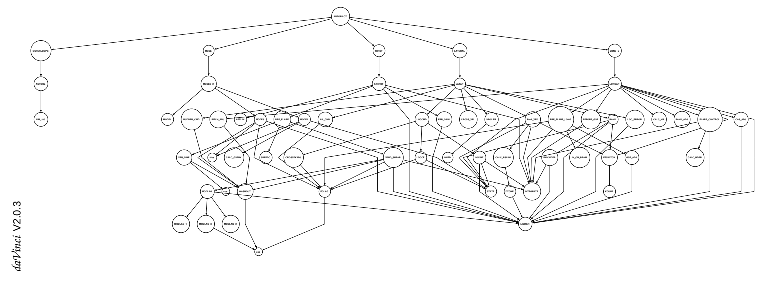

Figure 5: The function call graph of `b737` imparts some idea of its complexity as a program.

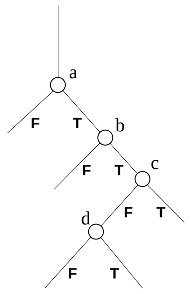

Figure 6: The flow of control for a hypothetical program. The nodes represent decisions, and the goal is to find an input that takes the TRUE branch of decision c.

Ten attempts are made with each method to obtain condition/decision coverage of `b737`. For the two genetic algorithms, we make some attempt to tune performance by adjusting the size of the population and the number of generations that can elapse without any improvement before the GAs give up. The goal of this fine-tuning is to maximize the percentage of conditions covered, while keeping the execution time low. For the standard genetic algorithm, we use populations of 100 inputs each, and allow 15 generations to elapse when no improvement in fitness is seen. For the differential GA we use populations of 25 inputs each, and allow 20 generations to elapse when there is no improvement in fitness (the smaller population size 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 gradient descent, we attempt to control the number of executions by varying the size of the increments and decrements in the input values, and we do the same for simulated annealing. We use randomly selected increments and decrements, chosen according to a Gaussian distribution with mean 4 and a standard deviation of 2. We use the same annealing schedule as in our other experiments.

Figure 7 shows two convergence graphs comparing random test generation, the standard GA, and the differential GA on the left, with gradient descent and simulated annealing on the right. The graphs show the pointwise best, worst, and mean performance over ten separate runs of each system. For each point on the horizontal axis, the *worst* curve shows the worst performance of the optimizer over all runs at that particular time. Thus, the performance never dropped below this line on any run. Likewise, the performance never rises above the curve marked *best*.

The best performance is seen on one of the ten standard GA runs. In this run, genetic search is able to achieve more than 93% CDC code coverage—significantly better than random test generation, which only achieves 55% coverage. Overall, the standard GA performs best, with a mean performance noticeably higher than that of the differential GA, and far above that of random test generation. There is little variability in the performance of the standard GA and almost none in the performance of the random test generator, but the differential GA exhibits surprising variability between runs. This may be caused by the smaller population we used for the differential GA; with fewer inputs in the population, it is less likely that statistical fluctuations in fitness will cancel each other out.

Because `b737` has a 186-dimensional input space, we had difficulty tuning gradient descent and simulated annealing so that they would only require 15,000 executions. Thus, the convergence graph on the right represents a different time period than the one on the left. However, we can simply cut off the executions of gradient descent and simulated annealing at 15,000 runs, making their results comparable to those obtained for the other three techniques. As we see from the figure on the right, simulated annealing in this case achieves about 91% coverage on average—slightly below the level of the standard GA. Gradient descent achieves about 82.5% coverage, somewhat below the other nonrandom techniques.

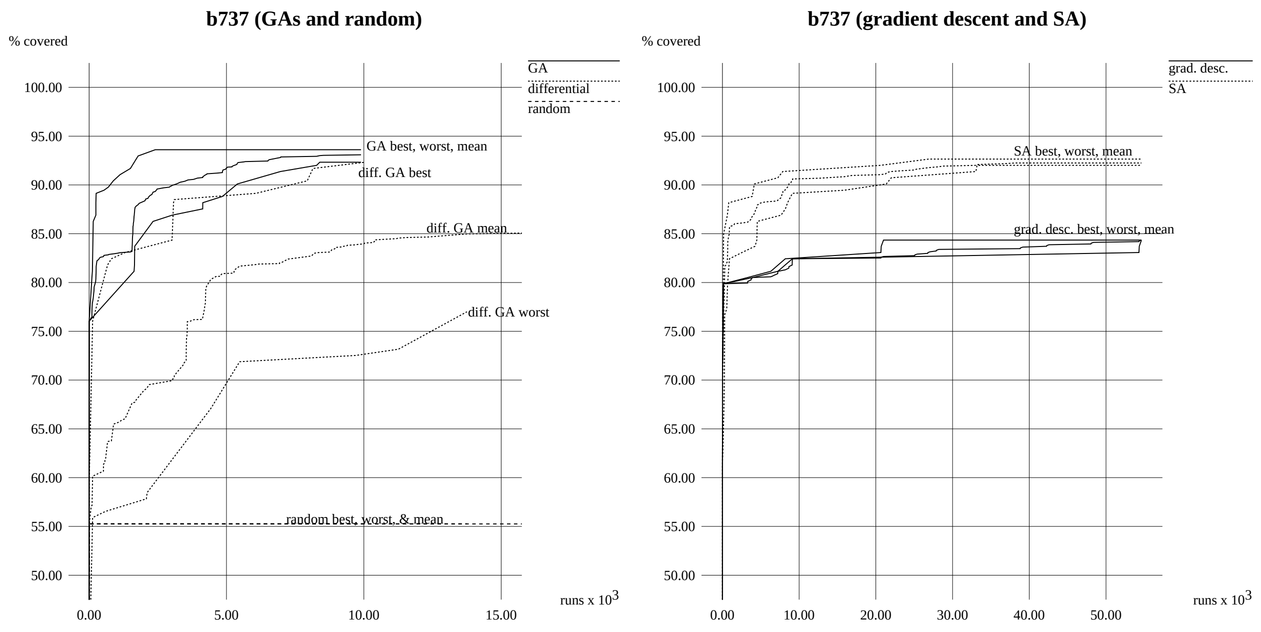

Figure 7: Convergence graph comparing performance of five systems on the **b737** code. Three curves are shown for each system. They represent the best performance, the pointwise mean over ten runs, and the worse performance. The GAs both show much better performance than random test data generation, with the standard GA generally outperforming differential GA. The performance of simulated annealing is just below that of the standard GA, and that of gradient descent slightly below that of the standard GA. Note the difference between execution times shown in the two plots.

Here, as in our other experiments, most useful inputs are discovered coincidentally. The fact that most inputs are discovered by luck means that most criteria *not* satisfied by chance were not satisfied at all. In this respect, the `b737` experiments shed some light on the true behavior of the test generators. We will examine a typical run of the standard GA and then discuss a run of 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 adequacy criteria on twelve different conditions in the code. Of these twelve attempts, only one was successful. The remaining eleven attempts show little forward progress during ten generations of evolution. While making these failed attempts, however, the GA coincidentally discovers fourteen tests that satisfy conditions other than the ones it was working on at the time. The high coverage level that is finally attained is mostly due to these fourteen inputs. Indeed, the most successful executions have the shortest run-times precisely because so many inputs are found coincidentally. Many conditions get covered by chance before the GA is ready to begin working on them. This does not mean that the GAs would fail to find those inputs if a concerted attempt had been made.

The GA failed to cover the following eight conditions, which we discuss further below:

```
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)
```

During a typical run of the *differential GA*, the 15982 executions of b737 involve 25 different attempts to satisfy specific criteria. Again, only one objective is obtained through evolution, though ten additional input cases, not necessarily prime objectives of the GA, are discovered.

The following are the conditions that the differential GA failed to cover:

```
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 by the GAs considered together 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 has limited effectiveness with such conditions; specifically, the handling of Boolean variables and enumerated types is currently inadequate [McGraw et al., 1997]. With an improved strategy for dealing with such conditionals, GA behavior should improve.

The GAs also fail 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 occur within decisions containing more than one condition, and this may account for the GA's 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. In almost all cases where the optimizers fail, they do so because all the inputs they examine lead to the same value for the objective function (causing a plateau effect).

4 Coincidental coverage

Our examination of the five test generation techniques uncovers a number of interesting features of the dynamic test generation problem. Perhaps the most significant of these is that test criteria are often satisfied coincidentally. That is, the test generator can find inputs that satisfy one criterion even though it is searching for inputs to satisfy a different one. The ability to find good tests by coincidence apparently plays a crucial role in the success of a dynamic test-data generation technique. This is in spite of the fact that random test generation, which should be good at finding inputs coincidentally, performs poorly in most of our experiments. The ability of combinatorial search algorithms to concentrate on a restricted part of the input space usually (but not always) gives them an advantage when it comes to discovering new inputs accidentally.

The two genetic algorithms were especially good at discovering inputs coincidentally, and it appears that this led to a sizable decrease in the computational resources they require. This is an interesting result, because the added sophistication of some combinatorial search algorithms slows them down, making them unsuitable for easy problems that could be solved by simpler techniques. In our experiments, coincidental discovery of inputs lets the genetic algorithms compete with simpler techniques on easy problems.

We also find that dynamic test generation frequently encounters plateaus in the objective function. These are regions where it is difficult or impossible for the algorithms to find inputs that change the objective function's value. Some of these plateaus are caused by `true/false`-valued variables appearing within conditions, but many of the plateaus cannot be explained in this way. It is somewhat surprising that plateaus should be encountered so often, especially when the search space has high dimensionality and allows the search algorithm to move in many directions.

4.1 Why random test generation breaks down

The most interesting question raised by our experiments 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, which ought to be good at finding things by chance, perform equally well?

Our explanation of this phenomenon is illustrated by the diagram in Figure 6, 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 will by definition take the FALSE branch at c, 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.

By contrast, inputs selected completely at random may be unlikely to reach d, because many will take the false branches of conditions a and b. Therefore randomly selected inputs are less likely to exercise new branches of d.

5 Conclusions

In this paper, we report on results from three sets of experiments using dynamic test data generation. Test data are generated for programs of various sizes, including some that are much more complex than those usually subjected to test data generation in the past. The following are some salient conclusions of our study:

- 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 adequacy criteria are intrinsically hard to cover, at least when condition/decision coverage is the goal. A criterion that is difficult to cover with one technique may often be easier with another. We are considering adaptive strategies to deal with this problem, including the ability to switch search strategies dynamically according to current search data.

- Our test generation techniques have a more difficult time with some programs than others. In particular, programs having deeply nested conditional statements and many conditions per decision are hard to cover using our techniques.

- Coincidental satisfaction of new test criteria can play an important role. In general, we found that the most successful attempts to generate test data do so by satisfying many criteria coincidentally. This coincidental discovery of solutions is facilitated by the fact that a test generator must solve a number of similar problems, and may lead to considerable differences between dynamic test data generation and other optimization problems.

- Plateaus in the objective function appear to be a common occurrence in dynamic test generation. Combinatorial optimization techniques tend to stall on such plateaus, and special techniques for dealing with them may useful. One promising technique is [Korel, 1990]'s dynamic path selection; other strategies are deserving of future research.

Our results show that automatically generating test data can be successfully accomplished using combinatorial optimization for complex programs written in C/C++. Though there are several remaining avenues to explore before the technology is fully mature, our initial experiments with GADGET are promising.

Acknowledgments

The authors wish to thank Michael Schatz and Berent Eskikaya for many helpful contributions to this paper. Tom O'Connor and Brian Sohr contributed the GADGET acronym. This research has been made possible by the National Science Foundation under award number DMI-9661393. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect those of the National Science Foundation.

References

[Chang et al., 1996] Chang, K., Cross, J., Carlisle, W., and Liao, S. (1996). A performance evaluation of heuristics-based test case generation methods for software branch coverage. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):585–608.

[Chilenski and Miller, 1994] Chilenski, J. and Miller, S. (1994). Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200.

[Duran and Natfos, 1984] Duran, J. and Natfos, S. (1984). An evaluation of random testing. *IEEE Transactions on Software Engineering*, pages 438–444.

[Gallagher and Narasimhan, 1997] Gallagher, M. J. and Narasimhan, V. L. (1997). Adtest: A test data generation suite for ada software systems. *IEEE TSE*, 23(8):473–484.

[Holland, 1975] Holland, J. (1975). *Adaption in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI. Reissued by MIT Press, 1992.

[Horgan et al., 1994] Horgan, J., London, S., and Lyu, M. (1994). Achieving software quality with testing coverage measures. *IEEE Computer*, 27(9):60–69.

[Kirkpatrick et al., 1983] Kirkpatrick, S., Gellat, C., and Vecchi, M. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

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

[Korel, 1996] Korel, B. (1996). Automated test data generation for programs with procedures. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 209–215. ACM Press.

[McGraw et al., 1997] McGraw, G., Michael, C., and Schatz, M. (1997). Generating software test data by evolution. Technical report, Reliable Software Technologies, Sterling, VA. Submitted to *IEEE Transactions on Software Engineering.*

[Michael et al., 1997] Michael, C., McGraw, G., Schatz, M., and Walton, C. (1997). Genetic algorithms for test data generation. In *Proceedings of the Twelfth IEEE International Automated Software Engineering Conference (ASE 97)*, pages 307–108, Tahoe, NV.

[Miller and Spooner, 1976] Miller, W. and Spooner, D. L. (1976). Automatic generation of floating point test data. *IEEE TSE*, SE-2(3):223–226.

[Storn, 1996] Storn, R. (1996). On the usage of differential evolution for function optimization. In *Proc. NAFIPS '96*, pages 519–523.

[Yin et al., 1997] Yin, H., Lebne-Dengel, Z., and Malaiya, Y. (1997). Automatic test generation using checkpoint encoding and antirandom testing. In *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE)*, pages 84–95.