Id: atcgp.tex version 1.1; 1997/05/23 13:05:32; Exp

Technical Report RSTR-003-97-11

Genetic Algorithms for Dynamic Test Data Generation

Christoph C. Michael, Gary E. McGraw, Michael A. Schatz, Curtis C. Walton

RST Corporation
Suite #250, 21515 Ridgetop Circle
Sterling, VA 20166

Version 1.1, May 23, 1997

Abstract

In software testing, it is often desirable to find test inputs that exercise specific program features. To find these inputs by hand is extremely time-consuming, especially when the software is complex. Therefore, numerous attempts have been made to automate the process.

Random test data generation consists of generating test inputs at random, in the hope that they will exercise the desired software features. Often, the desired inputs must satisfy complex constraints, and this makes a random approach seem unlikely to succeed. In contrast, combinatorial optimization techniques, such as those using genetic algorithms, are meant to solve difficult problems involving the simultaneous satisfaction of many constraints.

However, test data generation has only been applied to very simple programs in the past. Since they may not present great difficulties to random test data generation, it is difficult to compare the efficacy of different approaches when such programs are used as benchmarks.

In this paper, we discuss experiments with a test generation problem that is harder than the ones discussed in earlier literature — we use a larger program and more complex test adequacy criteria. We find a widening gap between a technique based on genetic algorithms and those based on random test generation.

This material is based upon work supported 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 author(s) and do not necessarily reflect those of the National Science Foundation.

1. Introduction

In software testing, one is often interested in judging how well a series of test inputs tests a piece of code — the goal is to uncover as many faults as possible with a potent set of tests. Thus, a test series that has the potential to uncover many faults is better than one that can only uncover a few.

Unfortunately, it is almost impossible to say quantitatively how many faults are potentially 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. This has lead to the development of *test adequacy criteria* — criteria that are believed to distinguish good test sets from bad ones.

Once a test adequacy criterion has been selected, the question that arises next is how one should 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 an obvious need for *automatic test data generation*.

In this paper, we report on an enhancement of a test data generation paradigm originally proposed by [Korel 90]. This paradigm treats parts of the program as functions that can be evaluated by executing the program, and whose value is minimal for those inputs that satisfy the adequacy criterion. Therefore, the problem of generating test data is reduced to the well-understood problem of function minimization. [Korel 90] proposed a gradient-descent algorithm to perform this minimization, but the problem is naturally amenable to combinatorial optimization techniques such as genetic search [Holland 75], simulated annealing [Kirkp. 83], or tabu search [Skorin. 90]. Our research involves the use of genetic search to solve the minimization problem.

In the past, it appears that automatic test data generation schemes were only applied to simple programs and simple test adequacy criteria. Random test generation has performed adequately on these problems. Nevertheless, it seems unlikely that a random approach could also perform well on realistic test-generation problems because these often require an intensive manual effort. Therefore, such simple experiments may have limited value when comparing sophisticated test generation schemes that are intended for real problems.

In this preliminary report, we use a larger program and more complex adequacy criteria, and compare genetic search with random test data generation. We find a widening gap between the two paradigms when our results are compared to earlier reports in [Chang 96] and [Korel 90]. Since our genetic algorithm has not yet been tuned for test data generation, this suggests that combinatorial optimization may, indeed, be useful for real problems where guesswork does not suffice.

2. Background

Empirical results seem to indicate that tests that are selected on the basis of test adequacy criteria are good at uncovering faults (see [Horgan 94], [Chilenski 94], [DeMillo 94b], for example). Furthermore, test adequacy criteria are objective criteria by which the quality of software tests can be judged. Neither of these benefits can be realized unless adequate test data (i.e., test data that satisfy the adequacy criteria) can be found.

The problem is that test data generation leads to an undecidable problem: one cannot take a test adequacy criterion and determine whether an input exists that satisfies it. The reason is that the problem of finding test-adequate is a variant of the halting problem — a problem not soluble by any algorithm.

To solve this dilemma, practical test data generation algorithms are usually given resource limitations (e.g., limitations on the amount of computation time they may expend). If no solution is found before the resources are exhausted then the algorithm is considered to have failed. In short, test generation algorithms 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 attain the greatest coverage with fixed resource limitations.

Clearly, 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 specifically addresses that need.

2.1. Test adequacy criteria

Most test adequacy criteria require certain features of a program's source code to be exercised. A simple example would be 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 tests*, because certain features of the source code are to be "covered" by the tests.

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

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

requires one program input that causes the value of the variable `a` to be greater than or equal to the value of `b`, and another that causes the value of `a` to be less than that of `b`. One effect of this requirement is to ensure that the "do one thing" and "do something else" sections of the program are both executed.

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. Somewhere between these extremes is *condition-decision coverage*, which 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, the question arises as to what to do when a test set fails to meet the chosen criterion. In most cases, the next step is to try to find a test set that *does* satisfy the criterion. It is often argued that tests sets that satisfy certain criteria are better at revealing faults. In addition, certain levels of test coverages are commonly required.

Since it can be quite difficult to manually search for test inputs satisfying certain criteria, test data generation algorithms are used to automate this process.

2.2. Existing test data generation methods

There are numerous existing paradigms for 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. We now briefly describe each of these.

2.2.1. Random test data generation

Random test data generation simply consists of generating inputs at random until a good 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 which case the number of adequate inputs may be quite small. Thus, the probability of selecting one by chance can be low.

This intuition is confirmed by empirical results. For example [Korel 96] 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.

2.2.2. Symbolic test data generation.

The majority of test data generation methods use symbolic execution to find inputs that satisfy an adequacy criterion. 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, in an ideal case, test case generation can be reduced to a problem of solving an algebraic expression.

However, a number of problems are encountered in practice when symbolic execution is used. One such situation arises when a condition controls the number of iterations of a loop. To obtain a complete picture of what the program does, it may be necessary 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 an an expression that yields an adequate test input, another path is tried.

However, there are other obstacles to a practical test data generation algorithm based on symbolic execution — loops are not the only programming constructs that cannot easily be evaluated symbolically. Problems can also 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 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 also be computed without the use of pointers or arrays, but it is not normal practice to avoid these constructs when writing a program. Therefore, 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 test class of test data generation paradigms is dynamic test data generation, exemplified by the TESTGEN system [Korel 90] [Korel 96]. This paradigm is centered on the idea that parts of a program can be treated as functions. One can execute the program until a certain location 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.

The function in question may not be easy to write down, but it can be *evaluated* for any input that causes the desired location to be reached. Furthermore, it is generally possible to find a function of this sort whose value will be minimal for those inputs that satisfy the adequacy criterion. Therefore, the problem of generating test data is reduced to the well-understood problem of function minimization: an input is sought that causes the function to assume its minimum value.

The search begins by establishing an overall goal, which is simply the satisfaction of the test adequacy criterion. 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 criterion is supposed to hold. If it does, then function minimization methods can be used to find an adequate input value (this will be described in greater detail below).

If the code is not reached, a subgoal is created to bring about the conditions necessary for function minimization to work. That is, 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 adequacy criteria. 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.

In our approach to test data generation, we adopt Korel's approach of converting the test generation problem to one of function minimization. This allows the problem to be solved by combinatorial optimization methods.

3. Genetic algorithms for automatic test data generation

This paper discusses the use of genetic search in test data generation. The framework is much the same as the one used in [Korel 90], because test data generation is reduced to function minimization in both cases. However, [Korel 90] uses a gradient descent algorithm to perform function minimization, and certain problems that have traditionally plagued gradient descent — especially the problem of local minima, which we will describe shortly — can be overcome by using genetic algorithms.

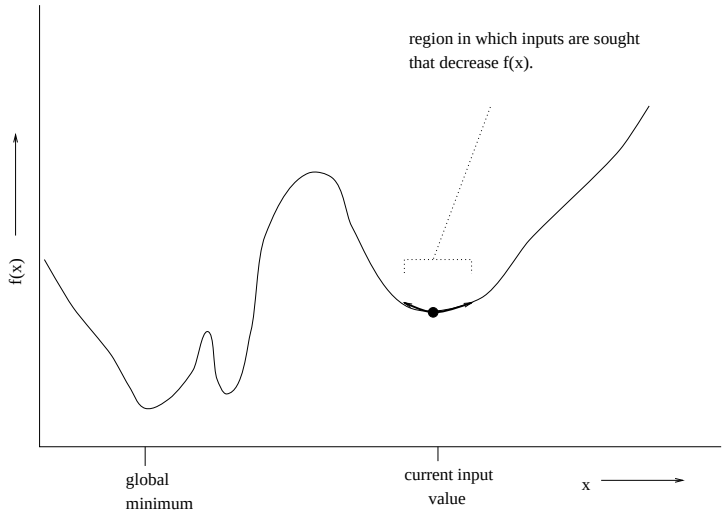

Figure 1: Illustration of a local minimum. An algorithm tries to find a value of x that will decrease $f(x)$, but there are no such values in the immediate neighborhood of the current x. The algorithm may falsely conclude that it has found a global minimum of f.

3.1. The issue of function minimization.

The ability to numerically minimize a function value is the key to dynamic test data generation. Various numeric computation algorithms are intended for this purpose (Newton's method, etc), but these require that the function to be minimized be relatively smooth. This may not be the case when the function is one computed by an arbitrary section of some computer program.

Therefore, more generic methods, making less stringent assumptions, are desirable. One of the simplest of these methods is *gradient descent*, which modifies the input values in small increments, and does so in such a way that the function's value always decreases. The process stops when no further decrease in the function value can be obtained by any of the modifications that are tried.

The minimization method suggested in [Korel 96] is a slightly enhanced 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 a constraint is violated (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.

Gradient descent and the method of [Korel 96] can both fail if a local minimum is encountered. A local minimum occurs when none of changes of input values that are being considered lead to a decrease in the function value, and yet that 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 Figure 1.

The problem of local minima has led to the development of heuristic function minimization methods, notably simulated annealing (see [Kirkp. 83]), tabu search (see [Skorin. 90]), and genetic algorithms (see [Holland 75]). We apply genetic algorithms to the problem of test data generation.

3.2. Genetic algorithms and function minimization

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. Both David Goldberg's book and Melanie Mitchell's book give thorough introductions to GAs and provide lists of possible application areas [Goldberg 89]; [Mitchell 96]. The beauty of GAs lies in their ability to model the robustness and flexibility of natural selection.

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. An initial population of individuals, each represented by a randomly generated genotype, is created. The GA starts to evolve good solutions from this initial population. The three basic genetic operators: selection, crossover, and mutation carry out the search. Genotypic strings can be thought of as points in a search space of possible solutions. They specify a phenotype which can be assigned a fitness value. Fitness values affect selection and in this way guide the search.

Selection of a string depends on its fitness relative to that of other strings in the population. The genetic search process is iterative: evaluating, selecting, and recombining strings in the population during each iteration (generation) until reaching some termination condition. 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 $P(t+1)$
    $t = t+1$

Evaluation of each string (individual) is based on a fitness function that is problem dependent. (In our particular case, the fitness function will be inversely related to minimality.) Determining fitness corresponds to the environmental determination of survivability in natural selection. Selection is done on the basis of relative fitness. It probabilistically culls from the population individuals having relatively low fitness. Mutation flips a bit with very low probability and insures against permanent loss of alleles. Crossover 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.

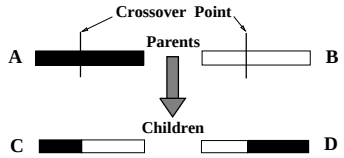

Figure 2: Single-point crossover of the two parents A and B produces the two children C and D. Each child consists of parts from both parents leading to information exchange.

Figure 2 shows how crossover mixes information from two parent strings, producing offspring made up of parts from both parents. This operator which does no table lookups or backtracking, is very efficient because of its simplicity. One-point crossover, where A and B are the two parents, producing two offspring C and D, is shown in Figure 2.

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 towards promising ones, without the GA having to *explicitly* remember its trail through the search space (Rawlins 1991).

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. Note that this search technique is not subject to the problems associated with local minima, which were described in section 3.1.

Since crossover causes genotypes to be cut and spliced, it is impractical to consider or track individual strings in analyzing a genetic algorithm. Instead, the analysis must focus on substrings which define regions of the search space and are called schemas. The schema theorem was introduced in the late 60's by John Holland [Holland 75]. Experimental evidence for the schema theorem is given in [Louis 93].

3.3. Test generation by genetic search.

In this section, we describe our approach to test data generation by genetic search. Recall 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. The two details that must be addressed are the same as in [Korel 90]'s paradigm: we must find a way to reach the code location where we want our test adequacy criterion to be satisfied, and we must convert that criterion into a function that can be minimized.

Recall that in the dynamic test generation paradigm of [Korel 90], 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.

When the desired code location is not reached, [Korel 90]'s paradigm uses backward chaining, as described in section 2.2.3. But it is also possible to use forward chaining: we can start from the beginning of the program, and look for inputs that take us down a path leading to the desired location.

Our approach is slightly different because our goal is (among other things) to cover *all* branches in a program. This means that all code locations should ultimately be reached (unless they are unreachable entirely) and we can simply delay our attempts to satisfy a certain condition until we have found tests that reach that condition.

This can be illustrated using the following code fragment:

```
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 if ((state_error < state_bndry[1]))
6:       state_weight = rule (state_error, state_bndry[3], state_bndry[1]);
```

In order to reach the `if` condition on line three, a test case must cause the condition on line one to be

false. The condition on line five can only be reached when the condition on line one and the condition on line three are both false.

If the goal of test generation is to exercise all branches in the code, then there will have to be test cases that cause the first condition to be true and test cases that cause it to be false. Therefore, when we have achieved branch coverage of line one, we will already have at least one test case that reaches the condition on line three. We are thus ready to begin looking for test cases that cover both branches of the decision on line three.

When we have covered the condition on line three, we will have test cases that reach line five, and therefore we will be ready to begin looking for test cases that exercise the branches of the condition on line five.

This leads to a test generation approach similar to the one employed by [Chang 96]. 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.

For the code fragment shown above, the situation is illustrated by table (a) in figure 3. In the table, the condition on line one has been covered because both the true and false branches have been exercised. Since the `false` branch of that condition was exercised, the condition on line three has been reached as well, and one of the two branches of that decision has necessarily been exercised. The table shows a situation where the condition on line three was also false, meaning that the condition on line five was reached and one of the branches of that condition has been exercised.

Figure 3: Two tables illustrating which conditions are covered in a program. Such tables can be used in deciding where to apply function minimization during automatic test data generation.

a

	T	F
line 1	x	x
line 3	-	x
line 5	-	x

b

T	F
x	x
-	x
-	-

In table (a), we may begin looking for a test case that exercises the `true` branch of the condition on line three, or one that exercises the `true` branch on line five. Both conditions are reached and so both are candidates for function minimization. Table (b) shows a different situation where line five was not reached because only the `true` branch was taken on line three. In that case, we cannot apply function minimization on line five because no test cases reach there. However, we can apply function minimization at line three. If we do so successfully, we will find a test that exercises the `false` branch of that condition, and that test will reach line five. After we have obtained one or such tests, the condition on line five will be reachable and we will be ready to begin function minimization there.

As we have already stated, we perform function minimization by using a genetic algorithm. We execute a new genetic minimization for each condition we reach. For example, in the case illustrated by figure 3(b) we would use the GA to cover the `false` branch on line three. With luck, we may find several tests to cover this branch, and it may even be that these tests cover both branches of the condition on line five — if this

happens, we will be finished. On the other hand, if our test cases only cover one of the branches on line five, we will have to run the GA again to cover the other branch.

When we run our genetic algorithm on a given condition, our initial population of test cases contains the tests reaching that condition. The initial population may also be contain some random test cases if the condition was not reached sufficiently often.

4. *Experimental results*

In this preliminary study, we used our test data generation algorithm on a single program: a component of a closed-loop high-temperature isostatic press controller that uses fuzzy logic to control the density of the metal in the press. The program contains 210 lines of executable `C` code and has 35 decision points. We attempted to generate test cases that satisfy *condition-decision coverage*: each condition in the code is required to be true in at least one test case and false in at least one test case, and in addition each control branch must be executed at least once. (A conditional statement may contain several conditions, and condition-decision coverage requires each of these conditions to take on both possible values. This is a stronger criterion than branch-adequacy, which only requires that the true and false branches of the conditional statement be executed.)

First, we tried to achieve condition-decision coverage with the genetic search algorithm just described. The program for which test data are being generated must be executed once each time the fitness function is evaluated for some test case, and we counted the number of these executions (there were 20,581 executions of the program).

Next, we applied random test data generation to the same program. We permitted the same number of program executions as was used by the genetic search — this amounts to generating 20,581 random tests, one for each time the fitness function was evaluated during genetic search.

Ultimately, we found that about 41% of the conditions in the program were covered by random test data generation. In contrast, genetic search covered 60% of the conditions.

This result differs markedly from those reported in [Chang 96]. In that paper, random test data generation was compared to several more sophisticated schemes on twenty-five small programs, containing an average of five decision points. In these experiments, only branch-adequate test cases were sought.

In [Chang 96], random test data generation covered 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.

Our results also differ from those reported in [Korel 96]. There, random test generation was compared to symbolic test generation and dynamic test data generation on three simple programs. Again, simple branch coverage was the goal. Random test generation achieved 67%, 68%, and 79% coverage, respectively, on the three programs used there. Symbolic test generation achieved 63%, 60%, and 90% coverage, while dynamic test generation achieved 100%, 99%, and 100% coverage.

The results of [Chang 96] and [Korel 96] are not directly comparable: [Korel 96] does not provide details about his symbolic execution algorithm, and it is unlikely to have been the same one as in [Chang 96]. In addition, [Chang 96] used different stopping rules for each of the test generation schemes, allowing some to consume more resources than others. [Korel 96] placed a common limit on the CPU time for all schemes.

Nonetheless, both sets of results show a common trend: random test generation has at least an adequate performance. In our own results, random test generation is far less adequate even though resource limitations were not a telling factor: the final 33% of the randomly generated tests failed to satisfy any new coverage criteria.

We attribute the poor performance of random test generation to the greater complexity of our code and the increased difficulty of attaining condition-decision coverage, as compared to branch coverage. Although earlier results suggested that random test generation might be quite valuable, our results indicate that this value may decrease considerably for complex programs and complex coverage criteria.

Genetic search, while outperforming random test generation by a considerable margin, did not perform as well as might have been hoped. However, our genetic algorithm is only in an early stage of development, and there are many avenues for further improvement. We have not yet implemented path selection heuristics, which were important in [Korel 96]'s work. We have not implemented an intelligent handling of boolean variables, which are currently treated like $\{0, 1\}$-valued variables instead of being treated as conditions at the point where they are defined. Finally, we have not yet tuned our genetic algorithm for the problem of test data generation, although past experience with GA's indicates that such tuning is usually needed.

5. Conclusion

In many areas of safety-critical software development, the ability to achieve test coverage of code is considered vital. This is especially true of aviation software development projects, because the Federal Aviation Administration currently mandates that safety-critical aviation software be tested with inputs that satisfy Multiple Condition Coverage. Test data is often generated by hand, so demand for automatic test data generation is high in these sectors.

There has also been an increase in the demand for coverage assessment tools elsewhere in the software development community. This may presage an increased demand for test data generation tools. No powerful test generation tools are commercially available today.

We have reported preliminary results from an experiment comparing random test data generation with a new approach using genetic search. Random test generation, which in earlier experiments seemed to be a viable approach, begins to look less promising in the more difficult setting we used for our experiment. Genetic search visibly outperformed random test generation in our small study, and although it did not perform as well as might have been hoped, the crude state of our current algorithm leaves many avenues by which this performance can be improved.

We think problems that are amenable to random test generation may not be the ones that are difficult for humans. Our results indicate that genetic search produces acceptable results for problems that are *not* amenable to random test generation, and that may therefore be more representative of the real problems where automatic test generation might be needed.

References

[Chilenski 94] Chilenski, J., Miller, S., "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal* pp. 193-200, Sept. 1994.

[DeMillo 94b] DeMillo, R. A., Mathur, A. P., "On the uses of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software," Purdue University Technical Report SERC-TR-92-P, 1991.

[Goldberg 89] Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, Massachusetts:Addison-Wesley, 1989.

[Chang 96] Chang, K. H., Cross, J. H. II, Carlisle, W. H, Liao, S-S., "A Performance Evaluation of Heuristics-Based Test Case Generation Methods for Software Branch Coverage," *International Journal of Software Engineering and Knowledge Engineering* Vol. 6, No. 4, pp. 585-608, 1966.

[Holland 75] Holland, J. H., *Adaptation in Natural and Artificial Systems*. University of Michigan, 1975.

[Horgan 94] Horgan, J., London, S., Lyu, M, "Achieving Software Quality with Testing Coverage Measures," *IEEE Computer* Vol. 27, No. 9, pp. 60-69, Sept. 1994.

[Kirkp. 83] Kirkpatrick, S., C. D. Gellat, Jr., Vecchi, M. P., "Optimization by Simulated Annealing," *Science* Vol. 220, No. 4598, pp. 671-680, May 13, 1983.

[Korel 90] Korel, B., "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering* Vol. 16, No. 8, pp. 870-879, August, 1990.

[Korel 96] Korel, B., "Automated Test Data Generation for Programs with Procedures," in *Proceedings of the 1996 International Symposium on Software Testing and Analysis*. ACM Press, pp. 209-215, 1996.

[Louis 93] Louis, S., McGraw, G. E., Wyckoff, R. O., "Case-based reasoning assisted explanation of genetic algorithm results," *J. Expt. Theor. Artificial Intelligence* Vol. 5, pp. 21-37, 1993.

[Mitchell 96] Mitchell, M., *An Introduction to Genetic Algorithms*. Cambridge, Massachusetts:MIT Press, 1996.

[Skorin. 90] Skorin-Kapov, J., "Tabu Search Applied to the Quadratic Assignment Problem," *ORSA Journal of Computing* Vol. 2, No. 1, pp. 33-41, Winter, 1990.