Genetic Algorithms for Dynamic Test Data Generation

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

RST Research
21515 Ridgetop Circle
Sterling, VA 20166
{ccmich, gem}@rstcorp.com

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, many 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.

In this paper, we discuss experiments with test generation problems that are harder than the ones discussed in earlier literature — we use larger programs 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.

1. Introduction

In software testing, one is often interested in judging how well a series of test inputs tests a piece of code — good testing means uncovering 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 [3]. 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. [3] proposed a gradient-descent algorithm to perform this minimization, but the problem is naturally amenable to combinatorial optimization techniques such as genetic search [1], simulated annealing [2], or tabu search [5]. Our research involves the use of genetic search to solve the minimization problem.

Real test-data generation problems involve the simultaneous satisfaction of many constraints, so the small programs that are traditionally used to benchmark test data generation algorithms may be too easy. We apply our technique to more complicated software, and find that the performance of random test data generation deteriorates. Genetic search performs considerably better.

2. Experimental results

In this preliminary study, we used our test data generation algorithm on two moderately-sized programs as well as a suite of small programs that have frequently been used as benchmarks for test data generation techniques. The first of the two larger programs are `fuzzy`, 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. The second is `b737`, part of an automatic pilot system, which has 69 decision points and 2046 source lines of code (excluding comments). 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

This material is based upon work supported by the National Science Foundation under award number DMI-9661393.

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.)

For each program, we attempted to achieve condition-decision coverage using genetic search, keeping a record of the number of times the program was executed (the program is executed each time the fitness function is evaluated for a new input). 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. In this way, we allowed random test generation the same computational resources that genetic search used.

For `fuzzy`, we found that about 41% of the conditions in the program were usually covered by random test data generation. By contrast, genetic search covered 60% of the conditions. For `b737`, genetic search acheived about 85% condition-decision coverage on average, while the random test-data generator consisently acheived just over 55%.

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.

We also compared random test data generation to genetic search on a library of small math programs, again with the goal of acheiving condition-decision coverage. The results are tabulated below.

Program	$random$	GA
Binary search	53.3	66.7
Bubble sort 1	100	100
Bubble sort 2	44.4	44.4
Number of days between two dates	35.3	39.2
Euclidean GCD	100	100
Insertion sort	100	100
Computing the median	100	100
Quadratic formula	75	75
Warshall's algorithm	91.7	100
Triangle classification	48.6	84.3

Genetic search outperformed random test data generation by a considerable margin in most of our experiments, and always performed at least as well. Still, it 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 [4]'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.

3. 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. It still remains to compare genetic search to techniques using gradient descent for test-data generation, such as the one used by [3].

References

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

[2] Kirkpatrick, S., C. D. Gellat, Jr., Vecchi, M. P.. Optimization by Simulated Annealing. *Science* (220)4598: 671-680, May 13, 1983.

[3] Korel, B.. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* (16)8: 870-879, August, 1990.

[4] 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.

[5] Skorin-Kapov, J.. Tabu Search Applied to the Quadratic Assignment Problem. *ORSA Journal of Computing* (2)1: 33-41, Winter, 1990.