Investigating the Performance of Genetic Algorithm-Based Software Test Case Generation

Donald J. Berndt and Alison Watkins
National Institute for Systems Test and Productivity
College of Business Administration, University of South Florida
dberndt@coba.usf.edu, awatkins@stpt.usf.edu

Abstract

Highly complex and interconnected systems may suffer from intermittent or transient failures that are particularly difficult to diagnose. This research focuses on the use of genetic algorithms for automatically generating large volumes of software test cases. In particular, the paper explores two fundamental strategies for improving the performance of genetic algorithm test case breeding for high volume testing. The first strategy seeks to avoid evaluating test cases against the real target system by using oracles or models. The second strategy involves improving the more costly components of genetic algorithms, such as fitness function calculations. Together, the various approaches offer opportunities for performance improvements that make these techniques more scalable for realistic applications.

1. Introduction

As software systems become more complicated and increasingly embedded in the processes of business and government, the costs of software failures become more severe. Complex distributed system failures are among the most troublesome problems, as the many components interact in unanticipated and unlikely ways. These highly interconnected systems often suffer from intermittent or transient failures that are particularly difficult to diagnose.

The research presented here is part of a stream focusing on the use of a genetic algorithm (GA) for automated test case generation. The aim is to allow large volumes of test data to be generated with a focus on any discovered errors. The genetic algorithm-based search process focuses on these rare or under-represented events, while still generating some test cases that explore the wider regions of successful executions. Many errors surface only after prolonged execution, with new and repeated test cases stressing the system. Our initial work used genetic algorithms to generate test cases for unit level testing. However, the real strength of large-scale automated testing would seem to be systems level testing [1]. How can genetic algorithms be used to *efficiently* generate large test data sets that are useful for uncovering failure patterns in complex systems?

2. Improving the Performance of Genetic Algorithms

There are two fundamental approaches for improving the performance of genetic algorithm-based software test case generation. One strategy is to use a system oracle, model, or simulation as a substitute for the actual system, thereby avoiding expensive execution costs for evaluating test cases. Predictive models can be based on a variety of technologies, including data mining approaches such as neural networks or decision trees. These predictive models may require some effort to construct, but the repetitive executions after deployment are typically very efficient. This can dramatically reduce the burden of evaluating test cases in each genetic algorithm generation. The following section will focus on one such approach, using neural networks as system oracles.

A second strategy involves direct efficiency improvements to the genetic algorithm. Genetic algorithms are iterative techniques that apply simple operations over and over again in the search for good solutions, or in this case, test cases. Each genetic algorithm generation includes many selection, crossover, and mutation operations. Any improvements to these component operations could mean significant gains in genetic algorithm execution times.

3. Neural Network-Based Oracles

In past experiments, the genetic algorithm was used to generate test cases for subsequent data mining efforts [1]. These efforts were aimed at using data mining to uncover failure patterns that might assist software maintenance and debugging activities. The genetic algorithm did produce good data sets for data mining since the search process focused on error regions, and therefore, generated more test cases for these under-represented or rare events. These same characteristics are likely to make the genetic algorithm suitable for generating training data for neural networks and other approaches that could be used as system oracles. As discussed above, system oracles can provide efficient model-based executions for performance improvements, replacing real target system executions.

To investigate the potential of using neural networks as system oracles, a genetic algorithm was used to generate test cases for training data, based on errors

Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04)
1530-2059/04 $20.00 © 2004 IEEE

injected into a distributed system test-bed. Two training sets of equal size were used, with the first set generated by the genetic algorithm and the second by random generation. A comparison of the two is shown in Table 1. The random-trained oracle was accurate 96% of the time, however, as there are far more non-error cases than errors, it is easy to achieve overall accuracy by predicting success. Therefore, a more appropriate comparison is based on predicting errors. In this situation, the performance of the random oracle deteriorates to 76% compared to 96% for the genetic algorithm-trained oracle. In fact, the random-trained network achieved reasonable performance on only two errors (hence the low per error accuracy), while the GA-trained oracle was able to identify errors in all but one case.

Table 1. Comparison of Accuracy for Random Versus GA-Based Test Cases

	GA	Random
Overall Accuracy	81%	96%
Error Accuracy	96%	76%
Average per Error Accuracy	83%	29%

4. Improving Fitness Function Calculations

Genetic algorithms are often used for optimization problems in which the evolution of a population is a search for a satisfactory solution given a set of constraints [2]. In this current application, the genetic algorithm is used to generate good test cases, but the notion of goodness or fitness depends on results from previous testing cycles stored in a fossil record. That is, a relative rather than absolute fitness function is used. The fossil record contains all previously generated test cases and information about the type of error generated, if any, and when that test case was run. Thus, the fossil record provides an environmental context for changing notions of fitness by comparing a test case to previously generated test cases and rewarding the individual based on fitness function terms such as novelty and proximity. However, computing against a constantly growing fossil record can be come a burdensome task.

A natural first step toward more efficient fitness calculations is to use a sample of the fossil record. This could lead to large reductions in computation time, as well as making the process predictable with fixed-sized samples. As long as the sample is fairly representative, the calculations should be acceptable approximations, providing adequate guidance for the search process.

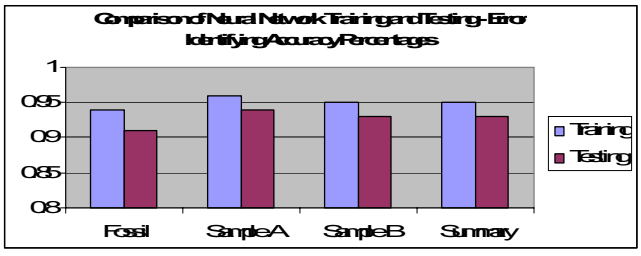

Figure 1. Accuracy for Sampling and Summarization

In order to investigate the performance of sampling, two samples, Sample A of size 500 (6%) and Sample B of size 5000 (17%), were used to calculate the fitness function in a set of genetic algorithm runs. Neural networks were then trained as a way to compare the overall quality of the data sets produced using the different levels of information in the fitness function. Figure 1 presents these results along with the original approach in which the entire fossil record is scanned, evaluating the neural network on training and unseen test data. Overall, sampling provides good search guidance while dramatically reducing the computation time. This simple strategy is easy to implement, adjustable with regard to sample size, and very effective.

Another approach is to summarize the fossil record forming a composite measure for groups of test cases, adopting a higher level of abstraction. In this way a smaller, fixed-size data structure could be used for fitness function calculations. Some of the advantages discussed above would be obtained, such as reduced and predictable computation times, but the strategy is somewhat complex and requires frequent re-calculation. Results for one "grid-based" summarization are also shown in Figure 1.

5. Conclusions

Several approaches for improving the performance of genetic algorithm-based test case generation are presented. The first approach uses system oracles or models to replace potentially expensive test case evaluations against real systems. The second group of techniques involves direct improvements to the genetic algorithm implementations. Two methods of speeding up costly fitness function calculations are explored, sampling and summarization. Sampling performed very well, with the added benefit of simplicity. These techniques make the use of genetic algorithm-based test case breeding possible on larger, more realistic projects.

6. References

[1] Berndt, D. J., A. Watkins, L. Johnson, K. Aebischer, and J. Fisher, "Using Genetic Algorithms and Decision Tree Induction to Classify Software Faults," *Proceedings of the Eighth INFORMS Conference on Information Systems and Technology* (CIST 2003), Atlanta, Georgia, October 18 – 19, 2003.

[2] Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Boston, Massachusetts, 1989.

Acknowledgement: This work was supported in part by the National Institute for Systems Test and Productivity, an interdisciplinary center at the University of South Florida under the USA Space and Naval Warfare Systems Command, N00039-01-1-2248.