View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AIS Electronic Library (AISeL)

Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2007 Proceedings

Americas Conference on Information Systems (AMCIS)

December 2007

Using Genetic Algorithms for Software Testing: Performance Improvement Techniques

James McCart
University of South Florida

Donald Berndt
University of South Florida

Alison Watkins
University of South Florida

Follow this and additional works at: http://aisel.aisnet.org/amcis2007

Recommended Citation

McCart, James; Berndt, Donald; and Watkins, Alison, "Using Genetic Algorithms for Software Testing: Performance Improvement Techniques" (2007). *AMCIS 2007 Proceedings*. 222.
http://aisel.aisnet.org/amcis2007/222

This material is brought to you by the Americas Conference on Information Systems (AMCIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in AMCIS 2007 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

USING GENETIC ALGORITHMS FOR SOFTWARE TESTING: PERFORMANCE IMPROVEMENT TECHNIQUES

James A. McCart, Donald J. Berndt, and Alison Watkins
Information Systems and Decision Sciences
College of Business Administration
University of South Florida
{jmccart, dberndt}@coba.usf.edu
awatkins@stpt.usf.edu

Abstract

One of the fundamental principles of agile software development involves delivering working software on a more frequent basis than traditional software development. To ensure the adequacy of these releases, testing must be undertaken which is capable of exploring and highlighting error-laden regions of software. Automated testing tools can provide support for quality assurance efforts within the context of rapid development cycles. Using genetic algorithms to generate test cases is one such method of automated testing for coverage, and more importantly, long-sequence testing. However, the execution time of such genetic algorithms may make their use somewhat unattractive. This research evaluates three performance techniques with the goal of decreasing execution time while still remaining effective in uncovering errors. Preliminary results suggest sampling provides dramatic efficiency improvements with several other summarization techniques improving performance even more.

Keywords: genetic algorithm, software testing, agile software development, functional testing

Introduction

Agile methods are lightweight methodologies that seek to control project risk through short iterative cycles, incremental delivery, self-organizing teams, emergent processes, and the active involvement of users to establish requirements and priorities (Boehm & Turner 2004). One of the fundamental principles of agile software development is to deliver working software on a frequent basis (Beck et al., 2001). Inherent within that process, software must be tested to validate conformance to relevant feature specifications. However, unlike more traditional development processes, which have infrequent releases and thus less frequent testing, the short release cycle in agile projects require testing on a much more frequent, if not continuous, basis. For instance, eXtreme Programming (XP) includes core activities focused on programmer-generated unit tests (prior to coding) and customer-centered functional testing of each feature (Beck 2000). While the time it takes developers to create test cases can be a burden, the tests are an important component of system documentation and build confidence with each successful feature deployment. Figure 1 depicts the programmer and customer testing perspectives, along with a third automated testing perspective. This research considers the addition of automated long-sequence testing within agile development methods. Automated test generation can enhance testing thoroughness with limited impact on programmer effort. In addition, automated testing is free of the cognitive bias (Stacey & Macmillian, 1995) that may restrict developers from fully exploring many testing regions of a program (or at least has a different bias). Therefore, automated means with which to create test suites can be beneficial in terms of time saved while creating test cases and also in discovering errors that may not have been uncovered until later in the development cycle.

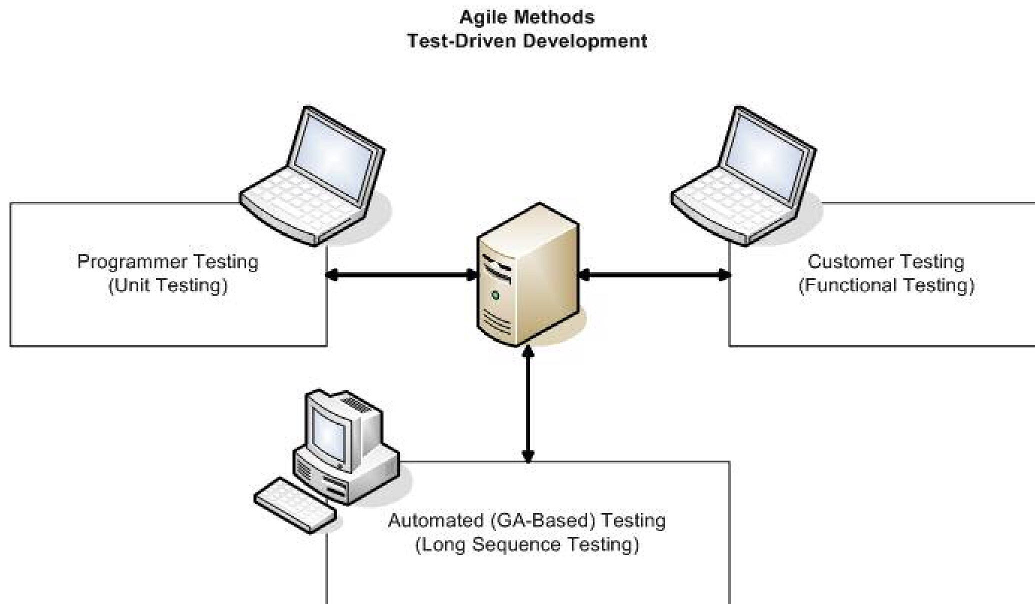

Figure 1: Adding Automated Testing to Agile Methods

One such approach is to automatically generate test cases using genetic algorithms (GA) (Berndt et al., 2003). GAs are capable of searching large areas of programs and uncovering error causing regions. In this case, the GA is used to pursue long sequence testing in which test cases (or close variants) are executed many times since failures may occur only after extended periods of execution. However, GAs can be restrictive in the amount of time they take to execute making them unattractive, especially when considered in the context of agile projects with shortened timetables that may require frequent testing. Therefore, the ability for GAs to not only be effective in uncovering errors, but also efficient is important, especially in agile projects where testing is valued but must be done quickly to fit in with the agile nature of the development process.

The problem this paper addresses is one of generating a test suite efficiently. A good test suite is one that consists of test cases that not only focus on error causing areas of the code, but provides a balanced representation of all errors. In addition, a test suite must include non-error generating test cases to confirm that the data actually tests all aspects of the software units. The speed of test suite generation is a cost issue – generating test cases by hand is costly and automatic generation is designed to reduce this cost. While GAs have been used before for test suite generation, our current purpose is to investigate ways in which test cases can be generated more rapidly by applying sampling or other performance enhancing techniques. In particular, this research evaluates different techniques of increasing the efficiency of GAs in terms of time required for execution, while still providing similar levels of coverage and error detection. More importantly, coverage is not the only testing concern. Researchers have noted that the same test case can run several times successfully and then fail due to some unique system state reached only after prolonged execution. Long sequence or high volume testing seeks to run a mixture of test cases, with repetition, to uncover such errors. GAs seem especially appropriate for such prolonged, yet automated testing in the context of agile programming.

This paper is structured as follows: first, we review the relevant literature on software testing in the context of GAs. Next, we briefly describe GAs and how they can be implemented for software testing. We then describe the different performance techniques used to test the efficiency of the GA. Finally, the testing environment is described along with the results and implications of the tests.

Software Testing and Genetic Algorithms

GAs have not been widely used in the software testing domain, instead they have typically been used to find solutions to optimization problems in many areas such as scheduling, game playing, and business modeling. Although not guaranteed to find optimal solutions, their strength lies in finding good solutions in relatively small amounts of time (Goldberg, 1989). The goal of GAs in this software testing research is to adequately test software as opposed to finding a single best solution in an optimization problem.

In the context of software testing, GAs have predominately been used to perform structural testing of programs. Structural

testing views a program as a white box, using source code in order to test the program. GAs have been used to examine branch (Khor & Grogono, 2004); block (Ma et al., 2005); and statement coverage (Pargas et al., 1999); branch testing (Wegener et al., 2001); boundary conditions (Tracey et al., 1998); and path testing (Watkins & Hufnagel, 2006; Borgelt, 1998; Watkins, 1995).

Functional testing, where the program is seen as a black box without the use of source code, has received much less attention. GAs have been used to generate test cases to stress test real-time systems (Briand et al., 2005); compared against the effectiveness of other techniques in terms of costs (Cai et al., 2006) and fault detection (Hu et al., 2005); and the results of GA generated test cases have been used in decision tree induction to create rules to help classify errors (Watkins et al., 2006). This research takes a functional approach to software testing by treating the program under test as a black box.

Genetic Algorithms

Genetic algorithms are an evolutionary computational method that mimics behaviors found in nature through mechanisms such as: selection, crossover, and mutation (Holland, 1975). A GA consists of a population, which is made up of a collection of chromosomes. Each chromosome in turn has a collection of genes which are digits encoded as a binary string. For example, below are two chromosomes with three genes represented as 36 bit binary strings. The first chromosome has gene values of 1956, 375, and 1043 while the second chromosome has gene values of 7, 1534, and 82.

```
011110100100 000101110111 010000010011       Chromosome 1
000000000111 010111111110 000001010010       Chromosome 2
```

Each chromosome is evaluated on its genes in a fitness function and assigned a fitness value, which determines the chromosome's probability of being selected for reproduction. When selected, two chromosomes reproduce and create offspring via crossover. A crossover operation involves selecting a random bit within a chromosome's binary string and swapping everything to the right of that bit with the other chromosome. To continue the example from above, if the 16th bit was selected as the crossover point, the first new chromosome would have gene values of 1956, 510, and 82 and the second new chromosome would have gene values of 7, 1399, and 1043.

```
011110100100 0001|11111110 000001010010   New Chromosome 1
000000000111 0101|01110111 010000010011   New Chromosome 2
```

The new chromosomes would also have a small probability of mutating during the reproductive process. Mutation is simply flipping a bit in a chromosome's binary string from 0 to 1 or vice versa. For example, if the second new chromosome's 3rd bit were mutated it would change from a 0 to a 1 which would result in its first gene value changing from 7 to 519. After the reproductive process, the two new chromosomes would be part of a new generation. This entire process of selection, crossover, and mutation continues with the goal of future generations having chromosomes with better fitness values due to being made up of characteristics of the stronger chromosomes in the previous generation.

Fitness Function

The decision of which chromosomes (test cases) are selected and bred for future generations is determine by the fitness function employed within a GA. In software testing the goal is to uncover errors while still ensuring adequate coverage of the search space. Thus, the fitness function must be able to accomplish that goal. Similar to Berndt et al. (2003), the value of our test cases is determined on the basis of their novelty and proximity. Novelty is defined as the uniqueness of a test case, either to all other test cases or within an error region, whereas proximity is how close a test case is to an error region. In order to determine if a test case is truly unique or close to error-causing test cases, the history of all test cases which have been run before must be kept and compared against. This record of history is kept in a chronological-like database called a fossil record (Berndt et al., 2003).

The fossil record contains all previously generated test cases and information about the type of error, if any, caused by each case. The benefit of having a historical record of prior test cases is to retain knowledge of which areas of the search space have been explored and what the result of that exploration was. Therefore, current generations can base their fitness function

off information gleaned from excavating the fossil record. For instance, determining the proximity of a test case to an error region helps to guide future test cases to that error region. However, once inside the error region the novelty of a test case ensures adequate coverage of the error region. In addition, the novelty of a test case against all other test cases will help ensure a good portion of the search space is examined and one area won't be overrun with a multitude of test cases if they don't cause errors.

In order to meet the goal of uncovering errors and providing adequate coverage of the search space a fitness function based off Berndt et al. (2003) and Watkins et al. (2004) was used (eq. 1). This fitness function is similar to Watkins (1995) and Borgelt (1998) in that it is dynamic since the fitness value assigned to a test case is dependent on what has occurred in previous generations. In equation 1 the fitness value is based off the Euclidean distance of a current test case compared to other test cases. Adjusting the weights and exponents in the fitness function changes the attention paid to exploring the search space or examining error regions. Equation 2 provides the formula to calculate the novelty of a test case against all test cases in the fossil record. Equation 3 provides the formulas to calculate the novelty and proximity of a test case against error-causing test cases in the fossil record. When the current test case has caused an error it is judged based off the novelty of the error within the error region, otherwise it is judged based off its proximity to the error region.

$$fitness = \left(\left(W_s * S^{1.5}\right)\left(W_r * R^{1.5}\right)\right)^2 \quad \text{(eq. 1)}$$

$$S = \sum_{i=0}^{F-1} \sqrt{\sum_{j=0}^{P-1} \frac{\left(c_j - f_{ij}\right)^2}{\max_j}} \quad \text{(eq. 2)}$$

$$R = \left\{ \begin{array}{ll} \sum_{i=0}^{E-1} \sqrt{\sum_{j=0}^{P-1} \frac{\left(c_j - e_{ij}\right)^2}{\max_j}} & if\ c = error \\ \frac{1}{\sum_{i=0}^{E-1} \sqrt{\sum_{j=0}^{P-1} \frac{\left(c_j - e_{ij}\right)^2}{\max_j}}} & if\ c \neq error \end{array} \right\} \quad \text{(eq. 3)}$$

Notation:

C – test case in the current generation, with c_j denoting the jth parameter of the test case
E – subset of f, which includes those test cases in the fossil record that were found to generate errors, with e_i denoting a specific test case
F – set of test cases in the fossil record, with f_i denoting an individual test case in the fossil record
max – set of maximum values for parameters, max_j is the maximum value for parameter j
P – set of parameters used for method invocation in a test case
R – novelty or proximity of a test case in an error region; formula used is dependent on if c generated an error
S – novelty of a test case in the search space
W – weight given to the search space (s) and the error region (r)

A disadvantage of GAs is their ability to get stuck in local optima and exhibit tunnel vision. This behavior is especially noticeable when the GA rewards test cases to fully explore an error region, which can lead to a disproportionate number of test cases focusing in on one error while ignoring the rest. In order to provide a more distributed number of test cases amongst errors, a Goldilocks mechanism (Watkins et al., 2004) was implemented. The purpose of the Goldilocks mechanism is to balance test cases amongst error regions by focusing test cases on barren error regions while removing focus from overly-saturated error regions. Figure 2 shows the contrast in distributions of errors between a GA without Goldilocks and one with Goldilocks. The one with Goldilocks shows a much more equitable distribution of errors and will be used for the remainder of this paper.

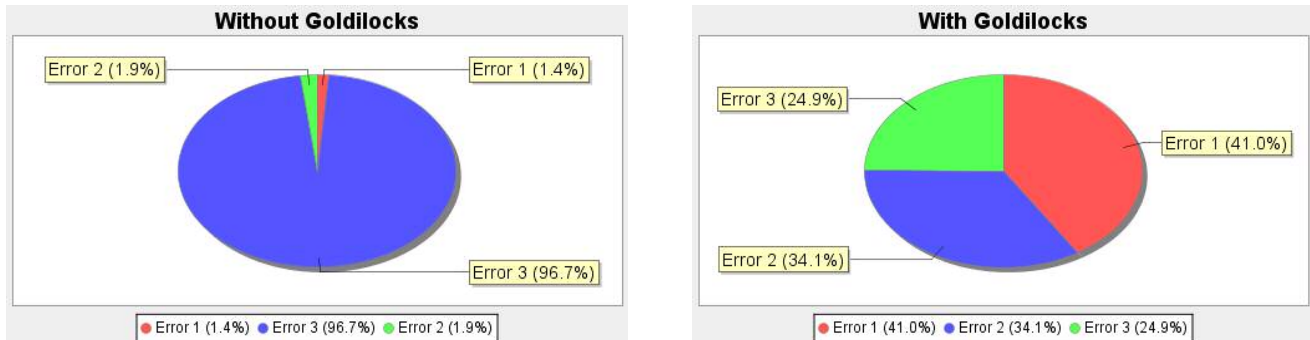

Figure 2. Error Distribution

Performance Techniques

Since the fitness functions are relative rather than absolute, they evolve with the population. Each generation is thus allowed to take advantage of insights gleaned from previous generations that are stored in the fossil record. However, computing fitness functions in terms such as novelty and proximity requires potentially costly distance calculations to each test case in the fossil record. Indeed, the computational burden grows as the fossil record grows. For instance, at the 5th generation only 1,600 distance calculations would be performed, while at the 500th generation 200,000 distance calculations would be performed. Improving this calculation is the most obvious path to better GA performance!

Sampling

As the size of the fossil record expands, the time to calculate successive future generations increases dramatically. Therefore, the use of the entire fossil record is not practical in situations with large populations, scarce execution time, or if the number of generations is large. Therefore, a simple way to decrease the computational burden while still retaining the rich history of the fossil record is to randomly sample test cases from the fossil record.

Taking a simple constant percentage of the fossil record will ensure the sample remains representative. However, the same computational problems arise, albeit later, than if the entire fossil record was used. Using fixed sized samples, on the other hand, keeps the number of distance calculations and thus the computational time constant, however in future generations the fixed-size sample will represent an increasingly smaller (and possibly less representative) proportion of the fossil record.

Sampling Frequency

Another issue with regard to sampling is the frequency of sampling. Although the number of distance calculations will be lower with sampling, and thus should take less time, sampling may introduce computational time when randomly selecting test cases from the fossil record. Therefore, changing the frequency of sampling may also have an impact on total time. However, sampling less often may cause newly uncovered information, such as a new error, to go unnoticed until a new sample is taken. Especially in later generations there may be very little change from generation to generation as most error regions are likely to have already been uncovered. In which case as long as the sample size is set large enough that sampling does not begin until most of the search space is explored, re-sampling less often than every generation may not result in much loss of information. Another possibility is to use some type of "progressive" sampling, with samples taken more often in the early stages and less often during later generations.

Error Representation

Error representation is an attempt by the GA to define the boundaries of each error. For each distinct error a representation of that error is generated, which is then used in distance calculations with all new test cases. Therefore, the number of distance calculations for the error region portion of the fitness function decreases dramatically compared with other methods. In order to define the boundary of an error, every new error causing test case is examined. If the new error-causing test case is outside the currently defined boundary of the error, then the boundary is extended to include the new error.

Origin Distance

Although dampened via sampling, the same costly aspect of comparing each new test case to many other test cases remains. Instead, the origin distance technique compares each test case to the origin of the search space and gets a vector distance (eq.

4). That vector distance is placed in a bin, which keeps count of the number of test cases with similar vector distances. The vector distances are considered similar based off the size of each bin (eq. 5). A fixed-size collection of bins is kept in distance sets, one for all test cases from the fossil record and the others for error test cases. To calculate novelty or proximity the vector distance of the current test cases are compared against the mid-point vector distance of each of the bins and multiplied by the number of test cases placed in that bin (eqs. 6 & 7). The same fitness function described previously (eq. 1) is used to calculate the fitness value.

$$D = \sqrt{\sum_{j=0}^{p-1} \left(\frac{c_j - \min_j}{\max_j} \right)^2} \quad \text{(eq. 4)}$$

$$Z = \frac{\sqrt{p+1}}{b-1} \quad \text{(eq. 5)}$$

$$S = \sum_{i=0}^{n-1} \left(\sum_{k=0}^{b-1} \left(|d_i - m_k| * f_k \right) \right) \quad \text{(eq. 6)}$$

$$R = \left\{ \begin{array}{ll} \sum_{i=0}^{n-1} \left(\sum_{k=0}^{b-1} \left(|d_i - m_k| * e_k \right) \right) & if \;\; c = error \\ \dfrac{1}{\sum_{i=0}^{n-1} \left(\sum_{k=0}^{b-1} \left(|d_i - m_k| * e_k \right) \right)} & if \;\; c \neq error \end{array} \right\} \quad \text{(eq. 7)}$$

Notation:

b – number of bins in an origin distance set
C – test case in the current generation, with c_j denoting the jth parameter of the test case
D – set of origin distances of test cases in the current generation, d_i is the distance for the ith test case in the set
e – set of counts in each bin of an error distance set, e_k is the count for the kth bin in an error distance set
f – set of counts in each bin of the fossil record distance set, f_k is the count for the kth bin in the fossil record distance set
m – set of origin distance mid-points in each bin, m_k is the origin distance mid-point for the kth bin
max – set of maximum values for parameters, max_j is the maximum value for parameter j
min – set of minimum values for parameters, min_j is the minimum value for parameter j
n – count of c test cases in the current generations
p – set of parameters used for method invocation in a test case
R – novelty or proximity of a test case in an error region; formula used is dependent on if c generated an error
S – novelty of a test case in the search space
Z – size of each bin in a distance set

Testing Environment

In order to evaluate the various performance improvement techniques, we utilized TRITYP, a commonly used program in software testing (Borgelt, 1998; Pargas et al., 1999; Watkins, 1995; Wegener et al., 2001). Taking the length of each side of a triangle as input (x, y, z), this program classifies a triangle as scalene, isosceles, equilateral, right angle or an invalid triangle. Our test bed allowed integer values from 0 to 2,000 as valid input to TRITYP resulting in a search space of eight billion unique combinations of input parameters. Three different errors were seeded into the program with each of the error regions consisting of 6.25% of the total search space. Figure 3 provides a graphical view of the error regions.

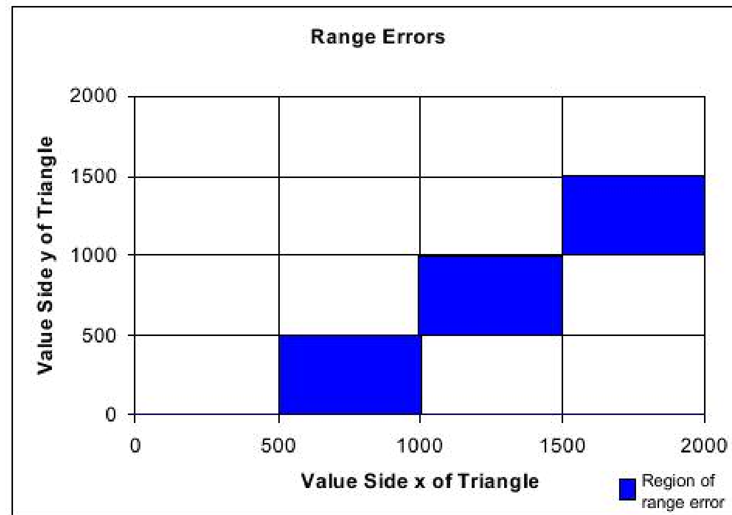

Figure 3. Range Errors (Berndt et al., 2003)

All of the performance techniques mentioned previously were tested. The fossil record was used as a baseline to compare all other results against as it uses the greatest amount of information regarding test case history. The goal of this research was to find performance techniques, which approach the fossil record's level of effectiveness, while improving on its efficiency. Total sample sizes of 250 (2.5%), 500 (5%), 1,000 (10%) and 2,000 (20%) were used to test sampling performance, with each part of the fitness function getting half the size of the total sample size (e.g. 125 for search space and 125 for error region for the 250 point sample). In addition, sampling frequency was tested at every generation and every ten generations to determine how timely sampling information needs to be. As the error representation is only concerned with error-causing test cases, a sample of 250 was used for the search space portion of the fitness function. The origin distance was run with a total of 500 bins per distance set. Finally, random test cases were generated, which did not use any mechanisms of the GA, for comparison purposes.

The results of the performance techniques were judged on three metrics, the first two dealt with the effectiveness in finding errors while the third was concerned with efficiency. The first metric, percentage of test cases that resulted in an error, demonstrated the ability of the performance technique at locating error regions. The ratio of rich-to-poor points was the second metric. A rich point is defined as any test case that results in an error or is a neighbor to an error-causing region, whereas a poor point is every other point (Watkins et al., 2004). In our experiments, any test case within 5% of an error region is designated as a neighbor. Although, the percentage of test cases causing error is important, it doesn't provide the whole picture of being able to define the boundaries of an error-region. Test cases which are close to an error-region but they themselves do not cause an error are important in determining where an error-region stops and starts. Therefore, a higher ratio of rich-to-poor points is beneficial. For instance, a 3:1 ratio would indicate for every one poor point, there are three rich points. This ratio is problem-dependent as many factors influence the distribution such as the number of distinct errors and the size of the search space. The last metric was the execution time to complete the task.

The GA was run 500 generations with 20 new test cases created each generation for a total of 10,000 test cases being generated and tested for each performance technique. All tests were performed independently on a 2.4 Ghz Pentium 4 computer with 512 MB of RAM running Windows XP with all of the results being averaged over 10 runs for each performance technique.

Results

Figure 3 provides a graphical representation of the X and Y parameters using the results from one of the 500 sample runs. It demonstrates how the emphasis of test cases shifts as the generations increase. As the generations progress, the density of test cases being run in the error-regions increase dramatically. This illustrates the ability of the GA to focus in on the error-laden regions. In addition, the number of errors amongst the different error-regions appears to be similar. This is due to the Goldilocks implementation mentioned previously. By helping spread the test cases more evenly, it assures one error-region doesn't get the lion's share of test cases.

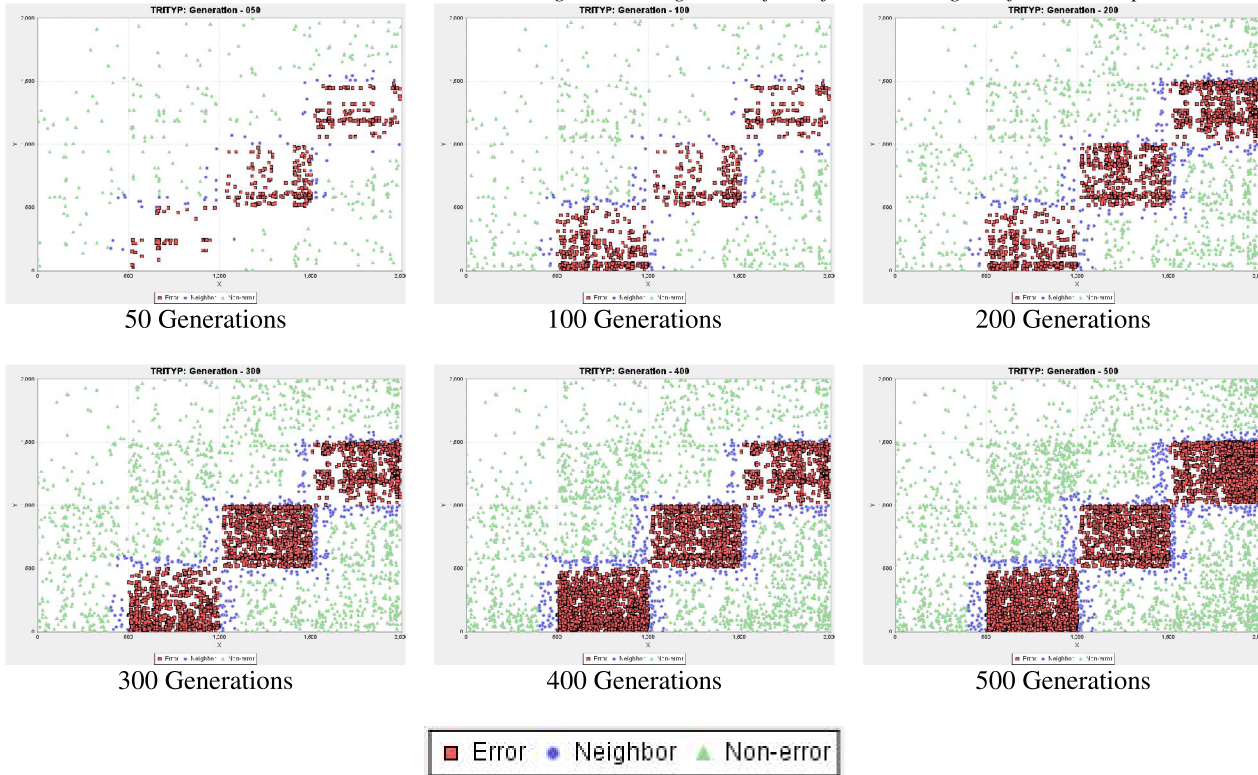

Figure 3. Graphical Results Over Generations

Table 1 presents the results of our experiments. Each performance technique is listed along with the averages and standard deviations of the 10 runs for the percentage of errors found, rich-to-poor point ratio, and the execution time to complete 500 generations. All performance techniques appear to be similar to the fossil record baseline in terms of average percentage of errors found and rich-to-poor point ratio (see figure 4). Between the performance techniques they differ at most by only 1.56% of errors and 0.22 in rich-to-poor ratio from the fossil record. The lack of substantial differences between the performance techniques on those two metrics demonstrates the effectiveness of uncovering and exploiting error-regions is quite similar between them.

Table 1. Performance Summary Metrics

Performance Technique*	% of Errors	Rich-to-Poor Point Ratio	Execution Time (Minutes)
Fossil Record	71.25% (0.92%)	3.52 (0.11)	64.33 (0.56)
Sample-250-1	70.25% (0.80%)	3.37 (0.09)	2.97 (0.02)
Sample-500-1	70.17% (0.85%)	3.39 (0.13)	5.30 (0.12)
Sample-1000-1	70.47% (1.21%)	3.37 (0.19)	9.89 (0.07)
Sample-2000-1	70.18% (1.14%)	3.45 (0.14)	17.96 (0.12)
Sample-250-10	70.20% (1.24%)	3.40 (0.17)	2.97 (0.01)
Sample-500-10	70.63% (1.79%)	3.46 (0.22)	5.36 (0.03)
Sample-1000-10	70.63% (1.22%)	3.43 (0.19)	9.82 (0.10)
Sample-2000-10	70.20% (1.10%)	3.39 (0.17)	17.77 (0.11)
Error Representation	70.50% (1.38%)	3.53 (0.16)	2.94 (0.04)
Origin Distance	69.07% (1.89%)	3.31 (0.32)	0.51 (0.00)
Random	18.78% (0.62%)	0.45 (0.01)	0.49 (0.01)
* Sample-x-y: x = total sample size; y = frequency of sampling			

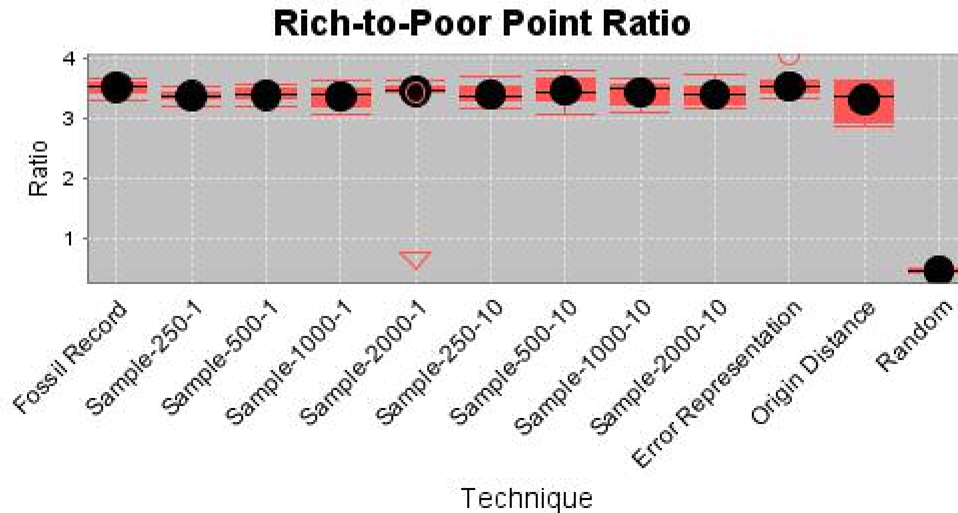

Figure 4. Rich-to-Poor Point Ratio

In terms of execution time there are large variations between the performance techniques. While any of the performance techniques realized a substantial improvement over the fossil record's average of 64 minutes, a difference of more than 17 minutes existed between the various performance techniques. As expected, the smaller sample sizes showed noticeable improvements in execution times compared with the larger sample sizes, ranging from about 3 to 18 minutes. There appeared to be no real efficiency gain from sampling less than every generation. The results of the error representation demonstrate the calculations required to represent the error regions did not decrease its efficiency as the execution time differed only seconds from the sample sizes of 250. The origin distance showed the most dramatic improvements in execution time by only taking an average of about 31 seconds to run, roughly one second longer than the randomly generated test cases.

Conclusion

Frequent software releases within agile projects require frequent testing to ascertain the correctness of the software being delivered. Using genetic algorithms to generate test cases that are free from cognitive bias can free developers from spending time writing test cases, allowing them more time to develop software. This research evaluated three different performance techniques used to increase the efficiency of a GA, while retaining the ability to uncover errors. All three performance improvement techniques performed well by not deviating strongly from the baseline fossil record in terms of percentage of errors found and the ratio of rich-to-poor points. In addition all three techniques demonstrated substantial execution time improvements over using the entire fossil record, even when using a sample size as large as 2,000.

The preliminary results of this research must be met with guarded optimism due to the fairly simple testing environment used. While this research is a first step in analyzing the performance of GAs utilizing a fossil record, further research is needed to examine how effective these performance techniques are in more complex testing environments. For instance, while the origin distance and error representation techniques worked well in our tests, it is unclear how they may work in situations where the errors are more complex or the search space is increased dramatically. The unidimensionality of the origin distance measure may fail to represent enough data in more complex environments, especially when numerous parameters are introduced, and thus may hinder a GA's ability to accurately determine the location of error regions and unexplored territory. In the same vein, error representation may also exhibit degradation when presented with errors that are not contiguous in nature. Therefore, performance techniques that are capable of summarizing the data in different forms may be required to handle these more complex environments, that is hybrid strategies that use sampling along with summaries, such as the origin distance technique, may achieve the greatest balance between efficiency, effectiveness, and ease of implementation.

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

References

Beck, K. (2000). *Extreme Programming Explained*. Boston, Massachusetts: Addison-Wesley.

Beck, K., M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas (2001). *Principles behind the agile manifesto.* Retrieved March 1, 2007, from http://www.agilemanifesto.org/principles.html .

Berndt, D. J., Fisher, J., Johnson, L., Pinglikar, J., & Watkins, A. (2003). Breeding software test cases with genetic algorithms. *Proceedings of the 36th Hawaii International Conference on System Sciences,* Big Island, Hawaii. 338-347.

Boehm, B and R. Turner. (2004). *Balancing Agility and Discipline*. Boston, Massachusetts: Addison-Wesley.

Borgelt, K. (1998). Software test data generation from a genetic algorithm. *Industrial Applications of Genetic Algorithms,* CRC Press.

Briand, L. C., Labiche, Y., & Shousha, M. (2005). Stress testing real-time systems with genetic algorithms. *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation,* Washington DC, USA. 1021-1028.

Cai, K., Li, Y., Ning, W., Wong, W. E., & Hu, H. (2006). Optimal and adaptive testing with cost constraints. *AST '06: Proceedings of the 2006 International Workshop on Automation of Software Test,* Shanghai, China. 71-77.

Goldberg, D. (1989). *Genetic algorithms in search optimization, and machine learning*. Boston, Massachusetts: Addison-Wesley.

Holland, J. (1975). *Adaption in natural and artificial systems*. Ann Arbor, Michigan: University of Michigan Press.

Hu, H., Wong, W. E., Chang-Hai, J., & Kai-Yuan, C. (2005). A case study of the recursive least squares estimation approach to adaptive testing for software components. *Fifth International Conference on Quality Software 2005,* 135-141.

Khor, S., & Grogono, P. (2004). Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. *2004 Proceedings 19th International Conference on Automated Software Engineering,* 346-349.

Ma, X., He, Z., Sheng, B., & Ye, C. (2005). A genetic algorithm for test-suite reduction. *2005 IEEE International Conference on Systems, Man and Cybernetics,* 133-139.

Pargas, R. P., Harrold, M. J., & Peck, R. (1999). Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability, 9*(4), 263-282.

Stacey, W., & Macmillian, J. (1995). Cognitive bias in software engineering. *Communications of the ACM, 38*(6), 57-63.

Tracey, N., Clark, J., & Mander, K. (1998). Automated program flaw finding using simulated annealing. *ACM SOGSOFT Symposium on Software Testing and Analysis (ISSTA 98),* Clearwater Beach, Florida, USA.

Watkins, A. (1995). The automatic generation of software test data using genetic algorithms. *Proceedings of the Fourth Software Quality Conference,* Dundee, Scotland. *2,* 300-309.

Watkins, A., Berndt, D., Aebischer, K., Fisher, J., & Johnson, L. (2004). Breeding software test cases for complex systems. *Proceedings of the 37th Hawaii International Conference on System Sciences,* Big Island, Hawaii.

Watkins, A., & Hufnagel, E.M. (2006). Evolutionary test data generation: A comparison of fitness functions. *Software: Practice and Experience*, 36(1), 95-116.

Watkins, A., Hufnagel, E. M., Berndt, D., & Johnson, L. (2006). Using genetic algorithms and decision tree induction to classify software failures. *International Journal of Software Engineering and Knowledge Engineering, 16*(2), 269-291.

Wegener, J., Baresel, A., & Sthamer, H. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology, 43*(14), 841-854.