International Journal of Applied Engineering and Technology ISSN: 2277-212X (Online)
An Online International Journal Available at http://www.cibtech.org/jet.htm
2012 Vol. 2 (4) October-December, pp.22-26/Tandon and Malik.

Research Article

BREEDING SOFTWARE TEST CASES WITH GENETIC ALGORITHMS

[*]Anisha Tandon and Poonam Malik
JIMS, Vasant Kunj, Delhi
**Author for Correspondence*

ABSTRACT
The genetic algorithm includes a fossil record that records past organisms, allowing any current fitness calculations to be influenced by past generations. This paper describes breeding software test cases using genetic algorithms as part of a software testing cycle. An evolving fitness function that relies on a fossil record of organisms results in search behavior, based on the concept of novelty, proximity and severity. It explores strategies that combine automated test suite generation techniques with high volume or long sequence testing. Long sequence testing repeats test cases many times, simulating extended execution intervals. These testing techniques have been found useful for uncovering errors resulting from component coordination problems, as well as system resource consumption (e.g. memory leaks) or corruption. Coupling automated test suite generation with long sequence testing could make this approach more scalable and effective in the field.

Key Words: *Coverage, Genetic and Test Case*

INTRODUCTION
Faulty software is costly in terms of money, public safety, and overall quality of life. As software systems become more complex and increasingly embedded in the processes of business and government, the costs of software failures continue to escalate. Software testing is an important area of research aimed at producing more reliable systems and controlling the costs of programming flaws. This paper focuses on using a genetic algorithm (GA) to breed software test cases. The technique compliments existing software testing procedures and is intended to be part of a software testing cycle as depicted in Figure 1. In this cycle, previous outcomes from any type of testing are stored in a data warehouse and used to evaluate the fitness of new GA-bred test cases. The new test cases are then run against the target system and the results are recorded in the historical data warehouse as the cycle continues.

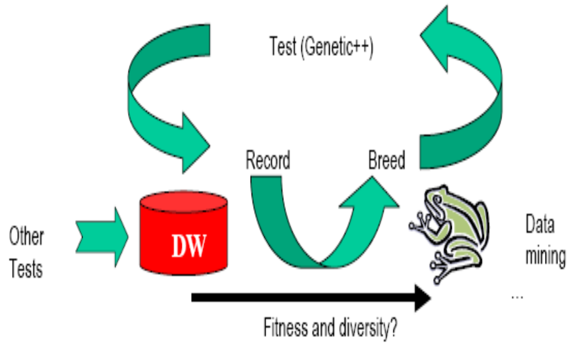

Figure 1: The Software Test Breeding Cycle

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 systems are particularly challenging, as the many components interact in unanticipated ways. These highly interconnected systems may exhibit intermittent or transient failures that are very difficult to diagnose and repair. Software testing is an important area of research and practice aimed at producing more reliable systems, as well as controlling the cost of programming errors. This paper focuses on high volume testing techniques that are especially appropriate for distributed or networked systems. Software testing techniques often involve developing test suites that achieve some level of "coverage" with regard to the target code.

Objective

While genetic algorithms have been applied to generating software test cases with some success, this research aims at extending this work in several innovative ways.

1. The software test cycle allows for incorporating test results from a variety of sources into test case breeding with genetic algorithms providing a powerful evolutionary and naturally parallel computational engine.
2. The approach seeks to balance fitness with diversity, so that a wide variety of test cases can be bred. In order to evaluate both measures, the GA framework is enhanced with a fossil record, which can be excavated to produce a highly diverse set of test cases.
3. The concepts of novelty, proximity, and severity are used to create a relative or changing fitness function. Analytic and visualization techniques are then developed for investigating and evaluating the resulting search behaviors in this non-optimization problem space.

Genetic Algorithms

A genetic algorithm (GA) is a search technique used in computing to find exact or approximate solutions to optimization and search problems. Genetic Algorithm were first introduced by Holland in 1975 as adaptive search technique that mimics the processes of evolution to solve optimization problems when traditional methods are deemed too costly in processing time. For example, Gas and other search techniques have been used to find solution to many NP-complete problems and have been applied in many areas, from scheduling to game playing, and from business modeling to gas turbine design.

GA includes a class of adaptive searching techniques which are suitable for searching a discontinuous space. The elementary operations are reproduction, crossover and mutation. Genetic algorithms maintain a population of solutions rather than just one current solution. Therefore, the search is afforded many starting points, and the chance to sample more of the search space than local searches. The population is iteratively recombined and mutated to generate successive populations. There are implicit parallelism search characteristics in GA, the probability stall into local minima will decreased infinitely. GA has some disadvantages, e.g. the combined optimization problem for complexes structure with large search space, long search time, and premature convergence.

GA is suitable when an optimized function cannot be solved with more accurate deterministic methods. In a combinatorial optimization like this, it is impossible to prove that they found optimum is a global one or to determine how close to the optimal solution.

Key features that distinguish GAs from other search methods include:

A population of individuals where each individual represents a potential solution to the problem to be solved.

A fitness function which evaluates the utility of each individual as a solution.

A selection function which selects individuals for reproduction based on their fitness.

Idealized genetic operators which alter selected individuals to create new individuals for further testing. These operators, e.g. crossover and mutation, attempt to explore the search space without completely losing information (partial solutions) that is already found.

A GA operates on strings of digits called chromosomes, each digit that makes up the chromosome is called gene and a collection of such chromosome makes up a population. Each chromosome has a fitness value associated with and this fitness determines the probability of survival of an individual chromosome in the next generation. To use a genetic algorithm, you must represent a solution to your problem as a genome (or chromosome). The genetic algorithm then creates a population of solutions and applies genetic operators such as mutation and crossover to evolve the solutions in order to find the best one(s).

A basic algorithm for a GA is as follows:

1. Generate a random population of chromosomes.
2. Calculate the fitness of each member of the population.

3. While best chromosome (highest fitness) is below some threshold or the number of generations is below some limit.
a. Generate next population (reproduction) allowing greater chance of more fit population members to survive.
b. Crossover a percentage of population members to create two new members.
c. Mutate a percentage of genes in the population.
d. Recalculate fitness of the population.
4. Finish.

While the algorithm begins with a random population, it is the iterative process of reproduction, crossover, and mutation that gives the process a structure. A chromosome on its own may have a poor fit or low fitness, but it may contain some characteristics which when combined with another chromosome (crossover) or adjusted slightly (mutation) will create an offspring with a much higher fitness.

Genetic Algorithms in Software Testing

Software testing is an essential part of software development which guaranteed the validation and verification process of the software. In order to do so we must have to adopt the process of mapping the software for all its transition states and individually validating the output for a set of given input. For a given part of software we will be writing a set of test cases called test suites.

The purpose of structural testing is to attain some level of code coverage, such as boundary conditions, individual or combined statement traversal or path coverage. Structural testing procedures are white box techniques that require the target code to be analyzed. The application of genetic algorithms to software test data generation differs from the optimization problem where a single goal is sought. It is not an optimization problem in that a single goal is not sought; rather what is optimized is the ‘coverage’ of the code under test.

There are several potential advantages that come from combining automated test case generation techniques, such as genetic algorithms, with high volume testing.

Automated test case generation is free from human biases that can influence handcrafted test suites (or at least have different biases). Thus, a combination of approaches should produce more robust test case collections.

Automated techniques can generate test cases that focus on particular areas before, and even during long sequence tests, adapting to ongoing test results.

Genetic algorithms seem particularly appropriate since the test case breeding process recombines highly fit building blocks to form new cases. Therefore, large populations of related test cases can be constructed with subtle differences introduced through crossover and mutation. So rather than repeating the same test case over and over again, many slight variations may enter the mix.

Breeding Test Suites using Genetic Algorithms

The type of search behaviors are of utmost concern in considering genetic algorithms as a test case generation technique to be used in conjunction with long sequence testing. Several research efforts have found genetic algorithms to be useful in software testing applications

Our past research has shown genetic algorithm-based test case generation can be highly focused. The fitness function used to evaluate individual test cases provides strong guidance, allowing the search to be tightly focused on selected criteria. However, the fitness function criteria can also be relaxed for a more breadth-first, near random approach.

Therefore, genetic algorithms implement a powerful search process that can be tailored to the situation at hand. In the current applications, the genetic algorithm is used to generate good test cases, but the notion of goodness or fitness depends on results from previous testing cycles.

That is, a relative rather than absolute fitness function is used. For the generation of test cases, a relative fitness function changes along with the population, allowing subsequent generations to take advantage of any insights gleaned from past results stored in the fossil record. This 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 the concepts of novelty, proximity, and severity giving rise to interesting search behaviors over successive generations simple rules, complex behavior. Novelty is a measure of test case uniqueness, while proximity is a measure of nearby error cases.

Figure 2 depicts the search space for three seeded errors using the TRITYP program from previous experiments. The spikes show that the genetic algorithm-based search process concentrated on the discovered error regions, generating many variations of localized test cases.

The search behavior can be described using a framework based on explorers, prospectors, and miners. An explorer is a test case that is highly novel, irrespective of whether it results in success or failure. Essentially, explorers are the test cases that are spread across the lightly populated regions of the test space. Once an error is discovered, the fitness function encourages more through testing of the region. Prospectors are test cases that are still somewhat unique, but are also near newfound errors.

Therefore, both novelty and proximity combine in the fitness function to generate prospectors. As prospectors uncover additional errors, the fitness function will reward points that are simply near other errors. Miners characterize test cases that are generated to more fully probe an area in which errors have been previously located. As more test cases are generated, the novelty and therefore overall fitness falls, encouraging new areas to be explored.

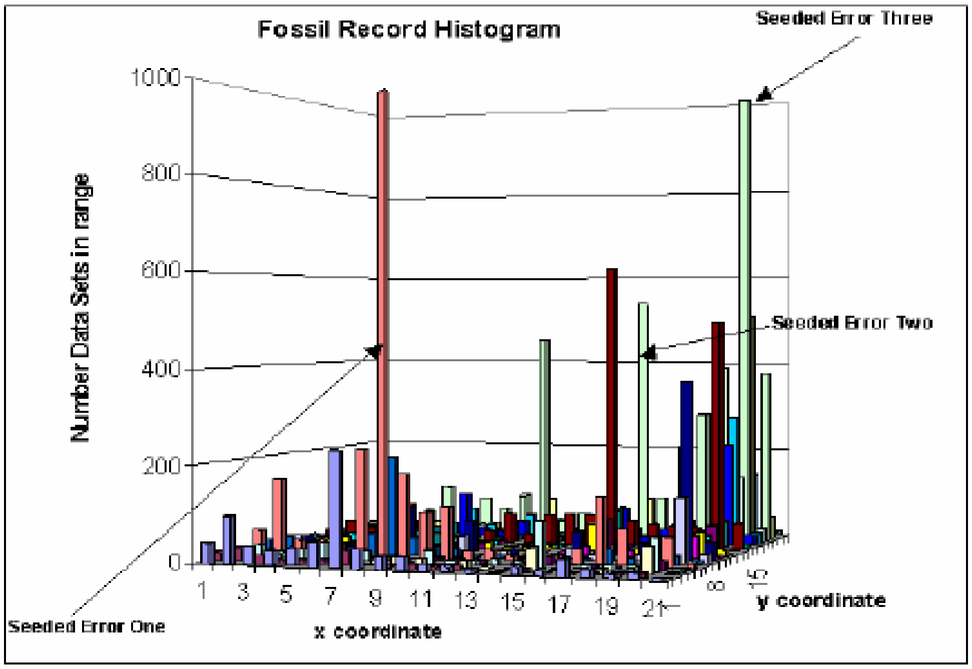

Figure 2: Visualization of a Fossil Record

In subsequent research, the testing search space was extended to include a collection of system environment attributes. The objective was to include factors that might be helpful in diagnosing transient errors in complex systems. For instance, identifies the types of failures that can occur when the distributed system fails to respond to a request (i.e. an omission failure), or the response is untimely.

Ghosh and Mathur highlight the importance of stress testing systems under varying load levels. Zhang and Cheung focus on stress testing multimedia systems, pointing out that network congestion and transmission errors or delays interrupt data delivery. Building upon these ideas, a set of system environment attributes was created, along with required input parameters for execution. The system attributes represent an important resource for identifying failure patterns. The basic set, including response time, elapsed execution time, cyclical temporal units (e.g. hour of the day or day of the week), tier load factors, and network load factors could obviously be extended in many ways, but provides an initial vocabulary to describe interesting patterns. Genetic algorithms proved to be useful search techniques in this greatly enlarged search space, uncovering and focusing on error-laden regions as before.

The system environment attributes, input parameters, and execution results represent an extended test case search space. Though the search space is very large, genetic algorithms have been shown to be a very effective search technique. Holland provided a theoretical basis for the power of genetic algorithms called schema theory, which showed that genetic algorithms derive much of their power from manipulating fit building blocks. The search problem at hand seems to be appropriate, with the system attributes and parameters providing building blocks that combine to partially identify failure patterns.

Genetic algorithms may have problems identifying isolated solutions surrounded by low fitness regions, basically a "needle in a haystack". In software failures, this would correspond to a very rare and complicated interaction of many attributes that causes a single failure. While these certainly occur, the bulk of software errors are likely to be less exotic and reasonable targets for genetic algorithm-based approaches.

CONCLUSION

It describes the preliminary results from a genetic algorithm based approach to software test case breeding.

The guiding fitness function can provide a focused search that produces a large number of localized test cases, or be loosened up for more random-like behaviors, depending on the testing scenario.

High volume or long sequence testing involves the repeated execution of a substantial number of test cases. Clearly, automated test case generation could be useful in building adequate test banks.

REFERENCES

Berndt DJ, Fisher J, Johnson L, Pinglikar J and Watkins A (2003). Breeding Software Test Cases with Genetic Algorithms Information Systems and Decision Sciences.

Haichang Gao, Boqin Feng and Li Zhu (2005). A kind of SAAGA Hybrid Meta-heuristic Algorithm for the Automatic Test Data Generation.

Berndt DJ and Watkins A (2005). High Volume Software Testing using Genetic Algorithms, College of Business Administration, University of South Florida.