Generalized Extremal Optimization: a competitive algorithm for test data generation

Bruno T. de Abreu[1]*, **Eliane Martins**[1] , **Fabiano L. de Sousa**[2]

[1]Instituto de Computação – Universidade Estadual de Campinas
Av. Albert Einstein, 1251 – Caixa Postal 6176 – 13084-971 Campinas, SP

[2]Instituto Nacional de Pesquisas Espaciais
Av. dos Astronautas, 1758 – 12227-010 São José dos Campos, SP

brunotx@gmail.com, eliane@ic.unicamp.br, fabiano@dem.inpe.br

Resumo. *O teste de software é uma parte importante do processo de desenvolvimento de software, e automatizar a geração de dados de teste contribui para reduzir esforços de custo e tempo. Foi mostrado recentemente que os Algoritmos Evolutivos (AEs) como, por exemplo, os Algoritmos Genéticos (AGs), são ferramentas valiosas para gerar dados de teste. Este trabalho avalia o desempenho de um AE proposto recentemente, a Otimização Extrema Generalizada (GEO), na geração de dados para programas que possuem caminhos com laços. O desempenho do GEO foi comparado com o de um AG, e os resultados mostraram que o GEO exigiu muito menos esforço computacional, tanto na geração de dados quanto no ajuste interno dos parâmetros. Isto indica que o GEO é uma opção competitiva para automatizar a geração de dados.*

Abstract. *Software testing is an important part of the software development process, and automating test data generation contributes to reducing cost and time efforts. It has recently been shown that evolutionary algorithms (EAs), such as the Genetic Algorithms (GAs), are valuable tools for test data generation. This work assesses the performance of a recently proposed EA, the Generalized Extremal Optimization (GEO), on test data generation for programs that have paths with loops. Benchmark programs were used as study cases and GEO's performance was compared to the one of a GA. Results showed that using GEO required much less computational effort than GA on test data generation and also on internal parameter setting. These results indicate that GEO is an attractive option to be used for test data generation.*

1. Introduction

Software testing is an important part of the software development process that aims to reveal faults in a software under test (SUT). Among many activities that help improving software quality, testing is the most used to assure quality and reveal faults [Binder 2000], even though being expensive in terms of effort and cost. Every piece of software developed needs to be tested to assure a minimum quality to the customer, and there are many famous examples of problems or even disasters that happened due to the lack of testing

*Supported by the Brazilian Federal Revenue Service.

before software deployment [Wikipedia 2006]. Although testing is usually done manually in industrial application, its automation has been a burgeoning interest of many researchers [McMinn 2004].

Test data generation consists of generating inputs for the SUT in order to evaluate its internal (white-box testing) or external (black-box testing) behavior, and there are different approaches to automating test data generation [Michael et al. 2001]. Besides improving the quality degree of the software delivered to the customer, the automation also reduces costs and time efforts [Binder 2000]. However, there are also limitations to putting this into practice. For instance, test data generation can become an undecidable problem even for simple test criteria. Typically, path testing consists of two steps: (i) select a finite set of paths to be exercised; (ii) generate test data to execute the selected paths. For the first step, a criterion is necessary, since testing all execution paths in a program is generally impossible due to the existence of infeasible paths and loops. In this paper, our concern is step (ii), that is, given a set of paths, how do you generate test data to exercise them?

The use of search metaheuristics has been proposed as a promising way to generate test data for complex problems in a reasonable time. The testing problem is translated into an optimization one using a math function (also known as fitness function), and the algorithms are used to maximize or minimize it. One of the most used is the Genetic Algorithm (GA) [McMinn 2004], a metaheuristic based on the principles of Darwin's Theory of Evolution that makes a global search in the design space for feasible solutions. The GA belongs to a more general category known as Evolutionary Algorithms (EAs), which are methods based on principles and models of biological natural evolution [Eiben and Smith 2003].

Another EA that was recently proposed is the Generalized Extremal Optimization (GEO) [Sousa et al. 2003]. GEO was originally developed as an improvement of the Extremal Optimization (EO) method [Boettcher and Percus 2001], which was inspired by the evolutionary model of Bak-Sneppen [Bak and Sneppen 1993] and has been applied successfully to real optimum design problems [Galski et al. 2004, Sousa et al. 2004, Sousa et al. 2003]. Its main advantage in comparison to other stochastic algorithms is that it has only one free parameter to adjust, which eases the process of setting it to give its best performance in a given application.

This paper assesses the applicability of GEO in software testing for test data generation to cover paths with loops of a SUT, extending the work presented by Abreu et al [Abreu et al. 2006]. It also shows the importance of a tuning process in order to improve the performance of the metaheuristics. In order to achieve these objectives, seven well known benchmark programs were selected for the experiments, including five programs that have paths with loops. GEO was compared to the Simple Genetic Algorithm (SGA) using three criteria: the average path coverage, the number of program executions, and the time spent until the end of the test data generation process.

In the next Section, basic concepts of software testing are presented. Section 3 introduces the EAs, along with a short description of the GA used in this paper for performance comparison with GEO. Section 4 includes a brief description of some related works by other authors. In Section 5 GEO is introduced, while Section 6 describes the

automatic test data generation approach used in this work. In Section 7 the results are presented and discussed, followed by the conclusions and an outline of future work in Section 8.

2. Test Data Generation for Path Testing

There are three approaches to test data generation [Burnstein 2003]: random, symbolic execution, and dynamic. The random approach simply generates test data randomly. Even though it does not require much effort, the probability of finding particular test data to satisfy specific test requirements of a complex SUT is very small [Michael et al. 2001].

Symbolic execution consists of attributing symbolic values to the software variables in order to get a mathematical and abstract characterization of what the software does. Among its many problems, the systematic derivation and manipulation of algebraic expressions is computationally expensive, specially when applied to a high number of paths [Sthamer 1996]; loop-dependent or array-dependent variables, pointer references and function calls are also problems for this approach [Michael et al. 2001, Korel 1990].

The dynamic test data generation approach was introduced in 1976 by Miller and Spooner [Miller and Spooner 1976] and uses optimization techniques to generate test data. The SUT execution enables knowing whether or not the test data generated satisfied one or more given test requirements. All the information needed by the optimization algorithm can be extracted from the SUT with program instrumentation, which can be made with a simple *printf(...)* or *System.out.println(...)* instruction, for instance. These instructions are from C/C++ and Java, respectively. Basically they receive a variable as parameter and display its content.

The two most common approaches to software testing are the white box and black box techniques. The former requires explicit knowledge about the internal behavior of the SUT to select test data, while the latter does not. These techniques are also known as structural and functional, respectively [Beizer 1990].

Path testing is a white box testing technique whose goal is to search the design space for suitable test data to cover every possible path of a SUT [Sthamer 1996]. A path in a program can be understood as a set of ordered and cascaded statements and branches. This ordered relation makes the path testing coverage criterion stronger than both statement and branch testing. However, covering all the possible paths on a SUT can be an impossible or computationally impractical task, for several reasons: (i) the number of paths in a program is exponential to the number of branches in it [Binder 2000], making an extensive covering harder to obtain as its size grows; (ii) loops in the program, something that is very common, can lead to an infinite number of paths [Lin and Yeh 2001, Pargas et al. 1999, Sthamer 1996]; (iii) there are paths in the program that will never be executed with any test data. These paths are also called infeasible, and their existence is associated with logic intrinsic to the program.

Therefore, path testing usually involves the selection of a subset of paths of the SUT to be covered [Lin and Yeh 2001], and the problem of infinite paths due to the presence of loops inside the program can be avoided by limiting the number of iterations of each loop [Sthamer 1996].

According to Sthamer [Sthamer 1996], automating software test data generation

must consider two things: the test data generator tool and the test adequacy criterion. The former is an algorithm that generates test data, while the latter evaluates the test data generated. Test data is usually evaluated by a fitness function (FF) that gives a grade to each candidate. This grade refers to the quality of a test data, which is measured by the coverage of test requirements achieved with it.

3. Evolutionary Algorithms

Evolutionary Algorithms (EAs) are stochastic search methods based on principles and models of biological natural evolution. The main idea behind EAs is to evolve a population of individuals (solution candidates) in an environment (problem) through competition, reproduction and mutation, in such a way that the average fitness (quality) of the population increases towards the solution for the problem at hand. They do not require a previous customization in order to be applied to an optimization problem, and the evolutionary process of the solution candidates is stochastic and guided by the setting of adjustable parameters [Eiben and Smith 2003].

Nowadays, EAs have four main branches: i) Evolution Strategies (ES); ii) Evolutionary Programming (EP); iii) Genetic Algorithms (GAs); and iv) Genetic Programming (GP) [Eiben and Smith 2003]. The basic differences among them are the structure representation of individuals, selection mechanisms, and the use of reproduction and mutation operators.

3.1. Genetic Algorithms

The Genetic Algorithm (GA) was originally proposed by Holland in the middle 1970s [Holland 1975], but it was in the late 1980s that they started to be studied more and applied to an increasing number of problems in science and engineering [Eiben and Smith 2003].

In Holland's first proposal, the SGA, the design variables are encoded in a binary string and each string represents an individual (or chromosome) of the population, which is initialized randomly. The major components of the SGA are the internal processes of selection, crossover and mutation. It uses fitness proportional selection (also known as roulette wheel method), one-point (or single) crossover, and a bit-by-bit mutation. A detailed description of the SGA can be found in many references, for instance in [Goldberg 1989].

In the test data generation field, the use of GAs can be tracked back to the work of Pei et al [Pei et al. 1994]. Most papers about test data generation with GAs use the same internal processes of the SGA, with slight differences. For instance, Pargas et al [Pargas et al. 1999] represented the individuals using real numbers instead of a binary string. Mansour and Salame [Mansour and Salame 2004] used a different selection process that included fitness proportional selection as in the SGA. Lin and Yeh [Lin and Yeh 2001] applied two-point crossover instead of one-point, and Michael et al [Michael et al. 2001] used the SGA on their experiments. For this reason, the SGA was chosen to be compared to GEO in this work. Moreover, a comparison between our results and these authors´ was not possible due to the lack of information on those works, e.g. the design variables domain, which could lead to an inadequate comparison.

4. Related work

Among the many alternatives to test data generator tools, the most used is the GA. Other options include Tabu search [Díaz et al. 2003], Gradient Descent (GD) [Korel 1990], Simulated Annealing (SA) [Tracey et al. 1998] and random search, often called Random-Test (RT). There are also many options for test adequacy criteria: path coverage [Watkins and Hufnagel 2006, Abreu et al. 2005, Mansour and Salame 2004, Bueno and Jino 2002, Lin and Yeh 2001, Pei et al. 1994, Korel 1990], branch coverage [Pargas et al. 1999, Sthamer 1996], condition-decision coverage [Michael et al. 2001] and statement coverage [Pargas et al. 1999] are some examples. Here we will present a classical work about dynamic test data generation and recent research that deals specifically with path coverage.

A classical example of the dynamic test data generation approach is the work of Korel [Korel 1990]. His approach was based on actual execution of the SUT, function minimization methods, and dynamic data flow analysis. When some input is executed in the SUT, the program execution flow is monitored. If an undesirable execution flow at some branch is observed, then the GD technique is applied to this branch to find input data in order to transverse the selected path. The dynamic data flow analysis allows determining the most promising variables to be explored when searching for this input data.

Lin and Yeh [Lin and Yeh 2001] generated test data to cover paths of a simple program using a GA and RT. They also proposed NEHD[1], the fitness function (FF) used in this paper, which is described in the beginning of Section 6. Although they stated that NEHD could deal well with paths with loops, they did not evaluate this feature in their work.

Bueno and Jino [Bueno and Jino 2002] proposed a new FF for path testing, capable of taking into account both the number of common branches between two paths and the branch where a deviation occurs from the desired path. In order to do this, cost functions were associated with each branch in such a way that test data that got far from covering the branch were penalized. Their aim was to maximize the number of common branches between two paths and minimize test data penalties. They evaluated their approach using a GA and RT on six benchmark programs, and the results were very successful.

Mansour and Salame [Mansour and Salame 2004] compared a GA, Korel's approach [Korel 1990] and the SA, for path testing. Their approach uses a FF that, among other things, evaluates symbolically the right and left side of each branch through the path. Each branch will have a fitness value, and the total fitness will be the sum of the fitness value of each branch that belongs to the path. Their approach has the ability to generate test data near branch boundaries, increasing the chance of revealing faults [Clarke 1976], but it only works for numerically valued branches, and does not address the coverage of program paths with loops.

In a recent work [Abreu et al. 2005], GEO was used for generating test data in order to cover the paths of the simplified triangle program [Lin and Yeh 2001]. GEO was compared to a SGA and RT, and NEHD was used as the FF. Besides the unsuccessful

[1]Normalized Extended Hamming Distance.

performance of RT, the results showed that GEO was competitive with the SGA metaheuristic, and this paper extends this first assessment by applying GEO to cover paths that include loops.

Watkins and Hufnagel [Watkins and Hufnagel 2006] produced an interesting work. They made a series of experiments comparing the FFs developed for path testing: Inverse Path Probability (IPP), NEHD [Lin and Yeh 2001], Bueno and Jino [Bueno and Jino 2002] and Mansour and Salame's [Mansour and Salame 2004] FFs. IPP is a variation of a FF Watkins proposed in her thesis in the nineties, and it works by encouraging the diversity of paths in such a way that test data associated with paths that are covered fewer times will have a better fitness than the others. Moreover, they proposed two other FFs that merge some characteristics of NEHD and Bueno and Jino's FF. The evaluation was made using a GA and RT on one benchmark program, and a very large program with many paths and input variables.

5. The Generalized Extremal Optimization Algorithm

The Generalized Extremal Optimization algorithm (GEO) is a recently proposed metaheuristic devised to tackle complex optimization problems. It was conceived as a generalization of the Extremal Optimization method (EO) [Boettcher and Percus 2001], which enabled GEO to be applied directly to a broad class of nonlinear constrained optimization problems, with the presence of any combination of continuous, discrete and integer variables. Both EO and GEO are evolutionary algorithms, and their internal processes were inspired by the simplified evolutionary model of Bak-Sneppen [Bak and Sneppen 1993].

The design variables are encoded in a binary string, as in the SGA. However, GEO associates a fitness number to each bit of the string, also called a species, instead of associating it to the whole binary string, as in the SGA. Hence, as shown in Figure 1, in the SGA there is a population of n strings with m bits while in GEO there is only one string, with a population of m bits. Note that in GEO each bit is considered a species of a population of species, because a bit mutation (change from '0' to '1' or from '1' to '0') produces with the other bits of the string a different configuration, which is indeed a different solution to the problem being tackled. Therefore, a binary string with n bits will have n species; for instance, consider the binary string $\{000\}$. The first bit mutation will yield $\{\mathbf{1}00\}$, the second one will give $\{0\mathbf{1}0\}$ and the third, $\{00\mathbf{1}\}$. Note that every bit mutation resulted in a different configuration of bits, hence a different solution. In a real application, the number of bits necessary to represent each design variable depends on its desired range and precision.

In the first step of GEO, a binary string (which encodes the inputs of the SUT) with m bits (or species) is initialized randomly. When a bit is mutated, it produces with the other bits a new configuration, which is a different solution to the problem. So, in order to evaluate how good each solution is, each bit is mutated (only one at a time) and an adaptability value (called $\Delta V_i = fx_i - fx_{best}$, where fx is the fitness function value and $i = 1, ..., m$) is calculated and associated with it. After the ΔV calculation, the bit is returned to its initial state. In the next step, the bits are ranked according to their ΔV values, from 1 for the worst ΔV to m, for the best. The way the bits are ranked depends on the type of optimization problem being tackled. In minimization problems, the smallest ΔV occupies the first position in the rank, and vice-versa for maximization ones.

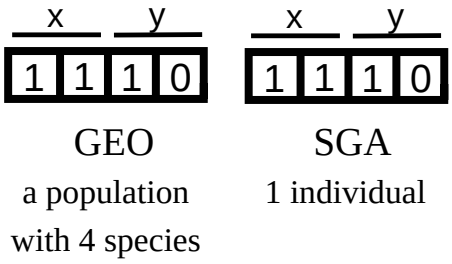

Figure 1. Representation of species and individuals in GEO and SGA. In the Figure, the variables x and y are encoded in a string represented by four species in GEO and by one individual in the SGA. Each string represents a solution for the problem.

After that, a bit is chosen with uniform probability to be a candidate to be mutated. Note that as a candidate, it may be mutated or not. The chosen bit will be mutated according to the following: a random real number RAN is generated in the range [0,1]; the mutation probability of the chosen bit i is calculated with equation $P(i) = k_i^{-\tau}$, where k_i is the rank position of i and τ, GEO's adjustable parameter; then the number RAN is compared to $P(i)$, and if $P(i) \geq RAN$, the bit i is mutated; otherwise, another bit is chosen to be a candidate and the process is repeated until a mutation happens. The algorithm loops through these steps until a stopping criterion is met, and the best configuration of bits found so far will be the solution.

In a variation of the canonical GEO described above, called GEO$_{var}$, the bits are ranked separately for each substring that encodes each design variable, and one bit for each variable is mutated at each iteration of the algorithm. The flowchart in Figure 2 shows the main characteristics of GEO and its variation, GEO$_{var}$.

The positive parameter τ is the single adjustable parameter of GEO. This gives GEO an a priori advantage over GAs, since they have more parameters to adjust. The value of τ influences how the search for feasible solutions is done. If $\tau \rightarrow \infty$ only the first bit in the rank will be mutated at each GEO iteration, and if $\tau \rightarrow 0$ any bit chosen (whether or not at a good position in the rank) will be mutated. According to previous works with GEO, the τ value that gives the best results generally lies in the range [0,10] [Abreu et al. 2005, Sousa et al. 2004, Sousa et al. 2003].

6. Implementation of GEO to Path Testing

This work uses the dynamic test data generation approach to generate test data to cover specific paths of a SUT. This approach was the best choice to accomplish the main objective of this work: the use and evaluation of GEO metaheuristic on test data generation for program with loops.

The path testing problem was modeled in a mathematical form using a FF specially developed to compare two paths of a program. This FF, called *Similarity* or NEHD [Lin and Yeh 2001], receives, as input, a target path (a path that must be covered) and another one, and quantifies the distance between them. The output is a number that shows the similarity between the paths and measures the quality of the test data[2]. The

[2]In this case, a test data has good quality if it results in the coverage of the target path.

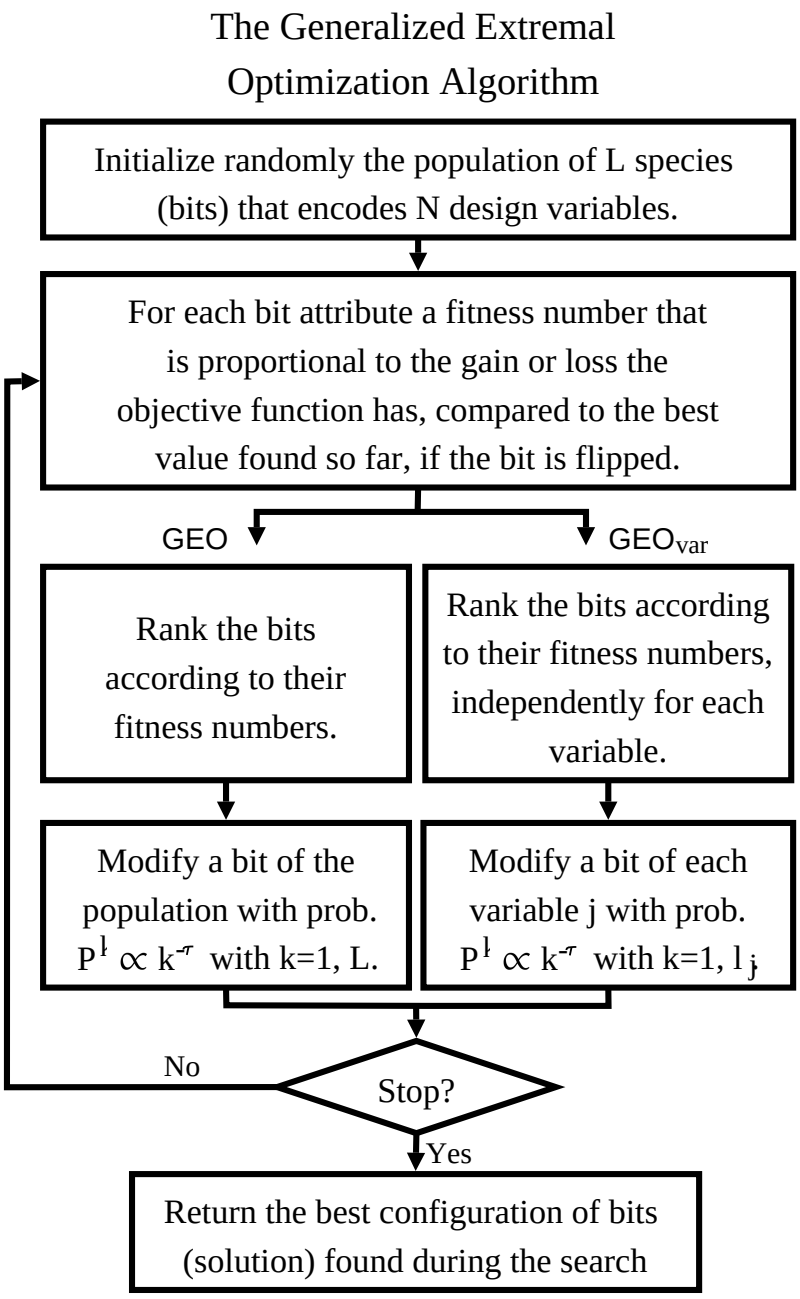

Figure 2. Flowchart of GEO and GEO$_{var}$, where l_j is the number of bits of each variable *j*, with $j = 1, N$ (from [Sousa et al. 2003]).

greater the similarity (NEHD's output), the better the test data.

Following the example given in [Watkins and Hufnagel 2006] to explain NEHD, consider that the target path is *abcd*. When test data generation begins, the first test data covers path *be*. The fitness of this test data will be calculated measuring the Hamming distance[3] from the first order to the nth order between the target path and the path covered. The first-order intersection set (IS) contains the branches that appear in only one of the paths; in this example, $\{a,c,d,e\}$. The first-order union set (US) contains the branches

[3]The number of different bits between two binary strings.

of both paths, $\{a,b,c,d,e\}$. Then, the first-order similarity will be one minus the size of the IS over the US, which is $1 - \frac{4}{5} = 0.2$. The second order involves all distinct pairs of contiguous branches. The distinct pairs of contiguous branches for the target path will be $\{ab,bc,cd\}$, and for the other path, $\{be\}$. The IS will be equal to the US, $\{ab,bc,cd,be\}$. Thus, the second-order similarity will be zero, and the set calculations stop. If the similarity was greater than zero, the process would continue with groups of three contiguous branches and so on until a similarity value of zero. Now each of the similarities will be multiplied by a weighting factor. The first-order factor is one, and the next will be the previous order factor multiplied by the number of distinct contiguous branches in the previous order set of the target path. In this case, the fitness will be $(0.2 * 1) + (0 * 4) = 0.2$. All the set operations and normalization processes are well detailed in [Lin and Yeh 2001].

NEHD was chosen for several reasons: first of all, it is a FF specially developed for path testing, meeting the requirements of this work; second, it does not take into account test data values that are closer to boundaries; even though boundary test data are more likely to reveal faults [Clarke 1976], this is not the focus of this work; third, although there are many set operations and calculations in the internal processes of NEHD, its implementation is very easy; fourth, it works at an abstract level that hides the SUT internal complexity; for instance, it does not matter if a branch condition within a path is too complex or not, it only considers that there is a branch, and will try to generate test data to cover it.

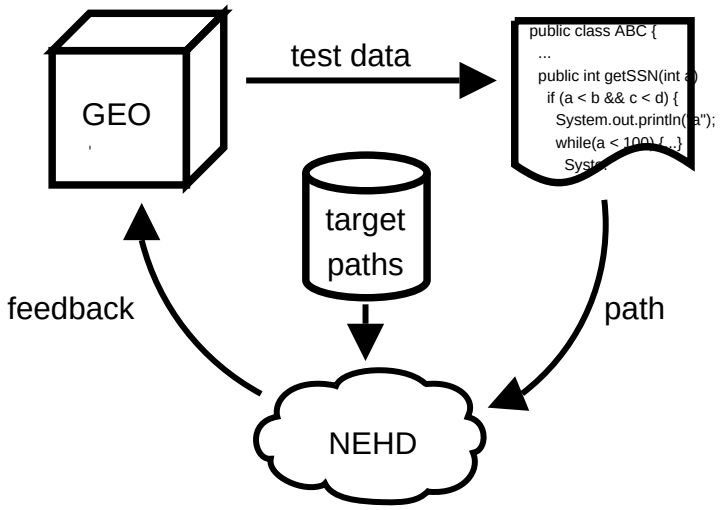

Figure 3. The dynamic approach used.

Figure 3 shows a high-level description of the test data generation procedure used. Given a set of target paths of a SUT which can be selected automatically or manually, one of them is selected randomly and then GEO starts generating test data trying to cover it. Each test data generated is input to the SUT, and the source code instrumentation previously inserted tracks which path was covered by the test data. The path covered is an output of the SUT and an input to NEHD, which calculates the fitness of the current test data. If the path covered is equal to the target path, it is removed from the set and stored in a list with the current test data. After that, another target path is chosen, and the test data generation process restarts. In order to improve coverage performance, if a

path covered is equal to any other target path of the set, it is also removed from the set and stored with its associated test data. This procedure loops until one or more stopping criteria are reached.

7. Results

The effectiveness of GEO was verified on seven Java subject programs (SPs) that were used by other authors [Watkins and Hufnagel 2006, Mansour and Salame 2004, Michael et al. 2001, Pargas et al. 1999, Sthamer 1996]: simplified triangle, remainder, product, linear search, binary search, middle value and triangle. All the experiments were run in an Intel XEON CPU with 2.40GHz, four processors and 1GB of RAM.

These SPs have different complexities and domains, as shown in Table 1. All the input variables are integers. SPs 2 through 5 have paths that include loops, and the paths with loops required zero, one, two and more than two iterations of the loop, as proposed by Sthamer [Sthamer 1996]. The domain (or range) of each input variable was set in order to make the search for feasible solutions more difficult, since smaller domains reduce the problem design space, increasing the chance of successful results of a random approach, for instance. In SPs 4 and 5 there is only one input variable, then only the key will be generated during each iteration of the search process. The array elements are initialized randomly in the beginning of each search process, and the array sizes used for SP 4 and SP 5 were 13 and 40, respectively.

Table 1. SPs characteristics.

#	SP	NIV[a]	Domain	NTP[b]	CC[c]
1	simp. triangle	3	[0,65535]	4	13
2	remainder	2	[0,65535]	5	2
3	product	2	[0,1023]	6	4
4	linear search	1	[0,16383]	5	4
5	binary search	1	[0,16383]	12	5
6	middle value	3	[-32768,32767]	4	6
7	triangle	3	[0,65535]	6	10

[a]Number of input variables.
[b]Number of target paths.
[c]McCabe cyclomatic complexity [McCabe 1976].

Although all SPs do not consider other variable types like string and boolean, the approach used here can handle this type of data. In order to do this, the test case designer would include a step between GEO and the SUT that properly converts the value generated by GEO into a char (remember that a string is an array of chars) or a boolean. A similar approach could be used to generate test data for more complex structures, e.g. an object. However, the generation of strings or meaningful objects is a problem when they must satisfy certain constraints for the input make sense [Michael et al. 2001]. Although this is major problem for real world testing, it will not be discussed here because it requires discussion in length.

In this work, GEO was compared to SGA using three criteria: the average percent of coverage achieved, the number of program executions and the time spent during all the test data generation process.

7.1. Tuning GEO and SGA

The performance of GEO and SGA may vary significantly as a function of the values of their adjustable parameters, so these algorithms were tuned for each of the SPs. GEO has only one parameter, τ, while the SGA has three: crossover (p_c) and mutation (p_m) probabilities, as well as population size ($popsize$).

The tuning was made by applying GEO and SGA in each SP of Table 1 one hundred times for each possible parameter combinations, with the number of NEHD evaluations set to 100000. GEO and GEO$_{var}$ started from the same random initial solution, and τ varied from 0 to 10 with increments of 0.25.The SGA $popsize$ parameter varied from 100 to 10000 with increments of $popsize * 10$; p_c varied from 0.6 to 1 with increments of 0.1; and p_m varied from 0.0010 to 0.0205 with increments of 0.0015. In real world testing, the test case designer could reduce the number of combinations and simulations in this step to save time, although this step should never be discarded.

Table 2. The best parameter combinations for each SP.

SP	GEO (τ)	GEO$_{var}$ (τ)	SGA ($popsize, p_c, p_m$)
1	3	7.75	1000, 0.7, 0.01
2	0.75	1	100, 0.9, 0.019
3	0.75	1.75	1000, 0.8, 0.0175
4	0	0	10000, 0.9, 0.0145
5	0.75	0.75	10000, 0.7, 0.01
6	2.5	3.5	100, 0.8, 0.0175
7	3.25	7.25	10000, 0.9, 0.019

GEO has fewer parameter combinations than SGA, and this makes its tuning process much less expensive than SGA's. This is well exemplified by noting that the tuning of GEO required executing the algorithm for 41 combinations of τ, while the SGA had to be run for 210 ($3 * 5 * 14$) combinations of its three parameters. Table 2 shows the best parameter combinations after the tuning for all SPs and algorithms. The many different combinations (mainly for the SGA) reinforce the importance of the tuning process.

7.2. Performance analysis

Using the combinations shown in Table 2 for GEO and SGA, the algorithms were executed on each SP two thousand times with the limit of 100000 NEHD evaluations (or SP executions), except for SPs 1 and 7. These SPs are the most complex (see Table 1), and had their limit set to 400000 and 800000, respectively, because they required a higher number of NEHD evaluations in order to better analyze their performance. The 2000 simulations were divided into 20 blocks of 100 simulations, hence results could be presented in a 0-100 scale. During each simulation, experimental data were collected every 100 SP executions. The stopping criteria used were the maximum number of NEHD evaluations, or the coverage of all target paths. The statistical test one-way ANOVA was used to analyze the results of coverage and number of SP executions; In those cases where ANOVA test indicated significant difference between the results, the protected F-test was applied to compare the algorithms.

Table 3 shows the results for the first evaluation criterion: the average path coverage and the average number of SP executions. The highlighted values indicate the best

results for each SP, and the presence of the * mark indicate that the difference between the results marked is not significant; for instance, Table 3 shows that the difference between SGA and GEO$_{var}$ for SP 1 is not significant, although there is a significant difference between SGA and GEO, and GEO$_{var}$ and GEO. The analysis will begin with SPs 2 through 6 because they are less complex than SPs 1 and 7. The average McCabe's cyclomatic complexity of those SPs is 4, and both of them have paths with loops. In SPs 2, 3 and 6, all algorithms achieved full coverage with just a few program executions. The results of GEO and SGA were similar, although GEO$_{var}$ executed SP 3 fewer times than SGA. Furthermore, even though SGA executed SPs 2 and 6 fewer times than GEO, the analysis of the second evaluation criterion shows that the time spent by GEO was very similar to SGA's.

Table 3. Average path coverage (top) and average number of SP executions (bottom).

SP	SGA	GEO	GEO$_{var}$
1	**92.41%***	89.19%	91.9%*
2	100%	100%	100%
3	100%	100%	100%
4	97.76%	**98.49%***	98.48%*
5	**99.83%**	99.53%*	99.52%*
6	100%	100%	100%
7	66.62%*	**66.67%***	65.58%*

SP	SGA	GEO	GEO$_{var}$
1	253640	268171	**226703**
2	**109**	651	419
3	1853	2277	**765**
4	**45343**	51721*	52229*
5	**48286**	51381*	50748*
6	**100**	550	267
7	799764*	**799300***	800000*

SPs 4 and 5 have array constructions besides paths with loops. Their complexities are almost the same of SPs 3 and 6, but the results showed that the algorithms executed the SPs a greater number of times when compared to SPs 2, 3 and 6, even using input variables with a domain 4 times smaller than those from SPs 2 and 6. The high number of SP executions indicates that test data generation for these SPs was more difficult than for SPs 2, 3 and 6. The average coverage for both algorithms was slightly different. In SP 4, GEO and GEO$_{var}$ were slightly better than SGA, and the opposite occurred in SP 5. Also note that the SGA population size for these SPs was 10000, which could explain why it executed fewer times the SPs; this number could give it an advantage since it starts the search from 10000 different solutions, increasing the chance of covering some paths right in the beginning of the test data generation process. However, even starting with only 14 species[4] (that is, 14 candidate solutions), a population 714 times smaller than SGA's, GEO and GEO$_{var}$ were better than SGA for SP 4, and almost as good as it for SP 5.

The next SPs, 1 and 7, have an average complexity 2.4 times greater than the

[4]This is the number of bits necessary to represent one input variable in the domain [0,16383].

other SPs. For this reason the coverage results of each one were also plotted in Figures 4 and 5. SP 1 is the simplified triangle program, and Figure 4 shows many interesting things: (i) all the algorithms covered 50% of the paths in the beginning of the search. Two paths (scalene triangle and invalid triangle) were very easy to cover, which explains this; (ii) The high τ value for this SP (see Table 2) indicates that this problem needs a more deterministic search, which cannot be done using a random approach. This explains the poor performance results of RT in [Abreu et al. 2005]; (iii) SGA got almost 75% of coverage in the first SP executions, and this is probably related to its initial population size, which is almost 21 times higher than GEO's.

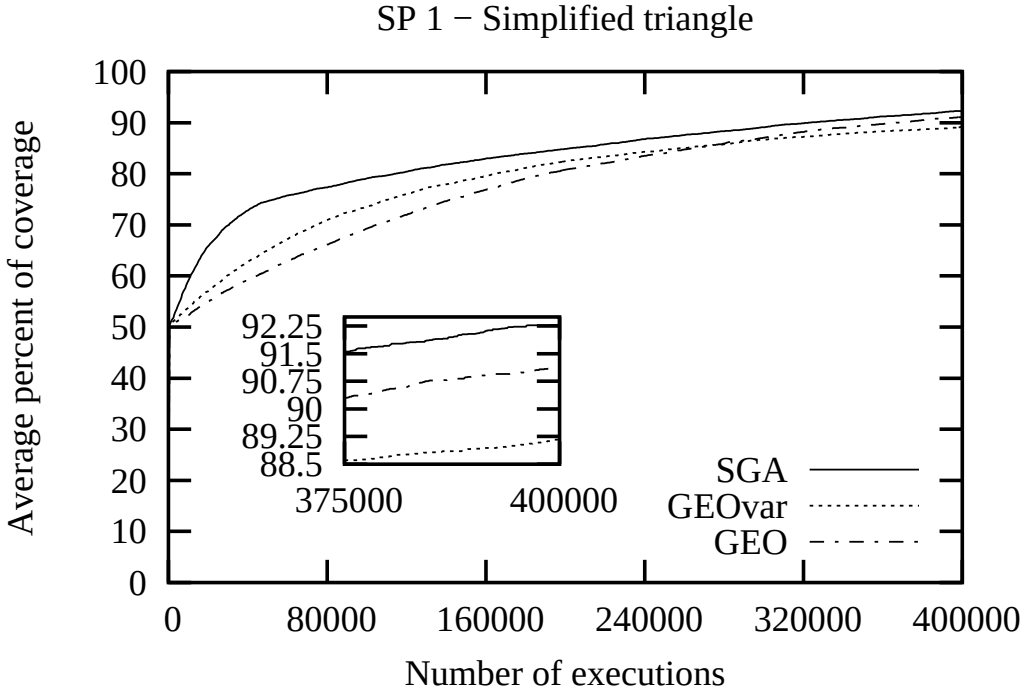

Figure 4. SP 1 behavior.

SP 7 was harder for all algorithms, as shown in Figure 5. Besides finding data to cover paths related to the same triangle types of SP 1, this SP classifies an scalene triangle in three categories according to its internal angles (right, acute and obtuse). Even allowing a high number of 800000 SP executions, no algorithm achieved full coverage or got even closer to 70%. There were some reasons for this: first of all, the paths not covered were the equilateral and right triangle ones. The equilateral triangle path had already shown to be difficult to cover on SP 1, but the right triangle was even harder. This triangle has an angle of 90 degrees, and its sides a, b and c need to satisfy the Pythagorean theorem $a^2 + b^2 = c^2$, where c is the largest side. This theorem is a nonlinear equality constraint, one of the most difficult types in optimization problems.

Another reason was that the set of target paths is implemented as a FIFO[5] queue, and for this particular SP the path corresponding to the right triangle preceded the equilateral one. Thus GEO and SGA were generating test data trying to cover the right triangle path first. However, besides being very difficult to find test data to cover it, the

[5]First in, first out.

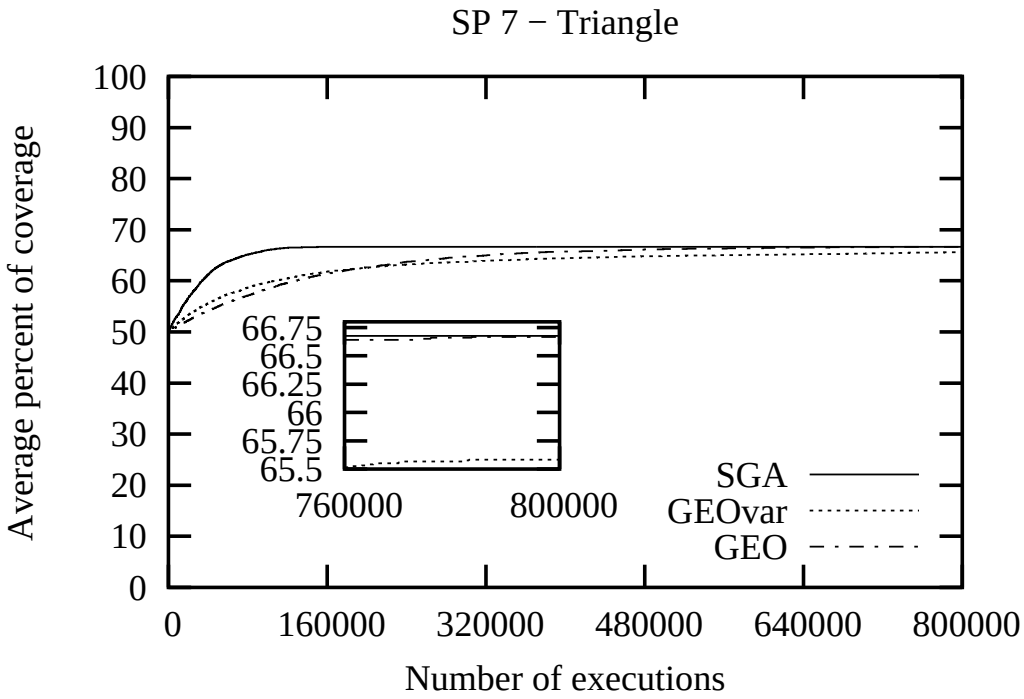

Figure 5. SP 7 behavior.

test data for this particular path are very different from those of the equilateral triangle, reducing dramatically the chance of serendipitous coverage. According to Michael et al [Michael et al. 2001], serendipitous coverage happens when the test data generator finds inputs that satisfy one test requirement even though it is searching for inputs to satisfy another one.

The second evaluation criterion was the time spent by the algorithms on 2000 simulations of each SP. Table 4 shows this information, and the highlighted items show the best results for each SP. The first thing that should be noted is that SP 7 spent the highest amount of time (varying from 44.67h to 67.34h), and one of the reasons for this was the maximum number of 800000 SP executions (or NEHD evaluations); almost every simulation executed the SP a 800000 times as a consequence of the extreme difficulty to find test data to cover the right triangle path. Both GEO and GEO$_{var}$ consumed much less time than SGA for this SP, and the final difference between them was almost 23h (almost one day).

One detail that indicates that GEO has a lower computational cost than the SGA is that the latter represents its population in $popsize$ arrays, each one with n bits, while the former represents its entire population in just only one array with n bits; for instance, in the case of SP 7 where the SGA population is 10000, SGA required $10000 * n$ bits to represent its initial solutions, while GEO used just n bits. This makes clear that GEO consumes less memory space than SGA. Besides that, SGA has in each of its three internal processes a random number generation step followed by a branch condition, two expensive operations. For instance, in the mutation process, these two operations are done for every bit of every individual in the population. In a population of 10000 individuals where each individual represents 3 input variables of 16 bits each, these operations will

be done 480000 times in just one execution of the mutation internal process.

Table 4. Time spent in the 2000 simulations for each SP.

SP	SGA	GEO	GEO_{var}
1	21.59h	16.19h	**16.05h**
2	**0.01h**	0.18h	0.22h
3	9.70h	2.58h	**0.82h**
4	14.10h	**12.51h**	13.99h
5	13.30h	13.53h	**10.16h**
6	**0h**	0.03h	0.01h
7	67.34h	**44.67h**	48.71h

In SP 1 the average coverage achieved by each algorithm was almost the same. However, GEO consumed less time (more than 5 hours) than SGA. According to Table 3, the average number of SP executions for SGA was 253640, while GEO executed the same SP 14531 times more. This result reinforces the previous observations made in the last paragraph.

The simplest SPs for test data generation were SPs 2 and 6: besides requiring less SP executions, the total coverage for all 2000 simulations was achieved extremely quickly. On these SPs, the worst result was of GEO_{var} on SP 2 where 2000 simulations took 13 minutes or, putting it differently, each simulation took an average of 0.396 seconds. SGA spent almost the same amount of time on SPs 4 and 5, even starting with an elevated population size — 10000 individuals —, which enabled right in the beginning of the test data generation process the evaluation of 10000 randomly generated solution candidates without executing any of the SGA internal processes.

There was an interesting result from SP 3. GEO_{var} spent much less time than SGA , being almost 12 times faster than it. Even though SGA executed the SP fewer times than GEO, the time it spent (due to its internal processes complexity) was 3.75 times higher than GEO's.

Briefly, the performance evaluation showed that GEO average coverage was very similar to SGA's in all SPs. The highest difference was of 3.22% on SP 1 between SGA and GEO. The SGA was faster only in SPs 2 and 6, and GEO and GEO_{var} took much less time than it to generate test data in the most complex SPs (1 and 7).

8. Conclusions and future work

This paper assessed the performance of a new evolutionary algorithm, the Generalized Extremal Optimization (GEO), on dynamic test data generation for program paths with loops. The performance of GEO was compared to the Simple Genetic Algorithm (SGA) in a set of seven benchmark programs (SPs). The performance criteria used were the average path coverage, which also included the average number of SP executions, and the time spent during the test data generation process.

Results showed that GEO can generate test data for program paths with loops in a competitive way to the SGA, while being much more easily tuned than the SGA to do this. The statistical tests did not show significant differences between SGA and GEO's results in all SPs (only in SPs 4 and 5) for coverage percent and number of SP

executions, but GEO spent significant less time than the SGA to generate test data for the most complex SPs. This result is important in the sense that for real world applications it is expected that the problems would be more complex than the benchmark programs used in this work. Another interesting point is that GEO or further derivatives of it, are also potentially competitive, in a general sense, to the GAs, since most of the GAs used so far to generate test data were very similar to the SGA.

In a follow up to this work, GEO will be compared to a more robust GA and another variation of GEO besides GEO$_{var}$. The use of another fitness functions to generate test data for path testing is also envisioned.

References

Abreu, B. T., Martins, E., and de Sousa, F. L. (2005). Automatic test data generation for path testing using a new stochastic algorithm. In *Proc. of the 19th Brazilian Symp. on Software Engineering*, volume 19, pages 247–262, Uberlândia, Brazil.

Abreu, B. T., Martins, E., and de Sousa, F. L. (2006). Generalized Extremal Optimization Applied to Path Testing. In *Supplementary proc. of the 17th IEEE Int. Symp. on Software Reliability Engineering*, volume 17, Raleigh, USA.

Bak, P. and Sneppen, K. (1993). Punctuated Equilibrium and Criticality in a Simple Model of Evolution. *Physical Review Letters*, 71(24):4083–4086.

Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition.

Binder, R. V. (2000). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1st edition.

Boettcher, S. and Percus, A. G. (2001). Optimization with Extremal Dynamics. *Physical Review Letters*, 86:5211–5214.

Bueno, P. M. S. and Jino, M. (2002). Automatic test data generation for program paths using genetic algorithms. *International Journal of Software Engineering and Knowledge Engineering*, 12(6):691–710.

Burnstein, I. (2003). *Practical Software Testing: A Process-oriented Approach*. Springer, 1st edition.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Trans. on Software Engineering*, 2(3):215–222.

Díaz, E., Tuya, J., and Blanco, R. (2003). Automated software testing using a metaheuristic technique based on tabu search. In *ASE*, pages 310–313.

Eiben, A. E. and Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer.

Galski, R. L., Sousa, F. L., Ramos, F. M., and Muraoka, I. (2004). Spacecraft Thermal Design with the Generalized Extremal Optimization Algorithm. In *Proc. of the Inverse Problems, Design and Optimization Symposium, Rio de Janeiro, RJ, Brazil, 2004, (in CDROM)*.

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

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press.

Korel, B. (1990). Automated Software Test Data Generation. *IEEE Trans. on Software Engineering*, 16(8):870–879.

Lin, J. and Yeh, P. (2001). Automatic test data generation for path testing using GAs. *Information Sciences*, 131(1-4):47–64.

Mansour, N. and Salame, M. (2004). Data Generation for Path Testing. *Software Quality Journal*, 12(2):121–136.

McCabe, T. J. (1976). A complexity measure. In *ICSE '76: Proc. of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA. IEEE Computer Society Press.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156.

Michael, C. C., McGraw, G., and Schatz, M. (2001). Generating Software Test Data by Evolution. *IEEE Trans. on Software Engineering*, 27(12):1085–1110.

Miller, W. and Spooner, D. L. (1976). Automatic Generation of Floating-Point Test Data. *IEEE Trans. on Software Engineering*, 2(3):223–226.

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

Pei, M., Goodman, E., Gao, Z., and Zhong, K. (1994). Automated Software Test Data Generation Using A Genetic Algorithm. Technical Report 6/2/1994, Michigan State University. Available at `http://www.egr.msu.edu/~pei/paper/GApaper94-02.ps`. Last access on 03/28/2006.

Sousa, F. L., Ramos, F. M., Paglione, P., and Girardi, R. M. (2003). New Stochastic Algorithm for Design Optimization. *AIAA Journal*, 41(9):1808–1818.

Sousa, F. L., Vlassov, V., and Ramos, F. M. (2004). Generalized Extremal Optimization: An application in Heat Pipe Design. *Applied Mathematical Modeling*, 28:911–931.

Sthamer, H. (1996). *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, Pontyprid, Wales, Great Britain.

Tracey, N., Clark, J., and Mander, K. (1998). Automated program flaw finding using simulated annealing. *SIGSOFT Softw. Eng. Notes*, 23(2):73–81.

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

Wikipedia (2006). Computer Bugs. Available at `http://en.wikipedia.org/wiki/Computer_bugs`. Last access on 01/14/2007.