Automatic test data generation for path testing using a new stochastic algorithm

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

bruno.abreu@ic.unicamp.br, eliane@ic.unicamp.br, fabiano@dem.inpe.br

Abstract. *Software testing is an important activity of the software development process, and automate test data generation contributes to reduce cost and time efforts. Path testing is a complex problem and metaheuristics have been proposed to deal with it. In this paper, an initial assessment of the efficacy of a recently proposed metaheuristic, the Generalized Extremal Optimization (GEO), in dealing with such kind of problem is made. Results show that it can be very competitive with the frequently used Genetic Algorithm (GA) metaheuristic, while requiring a much lesser effort in parameter setting, making it an alternative to such method in path testing.*
Keywords: Software testing, Automatic test data generation, Evolutionary algorithms, Path testing

Resumo. *O teste de software é uma atividade importante do processo de desenvolvimento de software, e automatizar a geração de dados de teste contribui para a redução de esforço de custo e tempo. O teste de caminhos é um problema complexo e metaheurísticas foram propostas para lidar com ele. Neste artigo é feita uma avaliação inicial da eficácia de uma metaheurística proposta recentemente, a Otimização Extrema Generalizada, no tratamento deste tipo de problema. Os resultados mostram que ela pode ser competitiva com a tão utilizada metaheurística chamada Algoritmo Genético, exigindo muito menos esforço na definição dos parâmetros e tornando-a uma alternativa a este método no teste de caminhos.*
Palavras-chave: Testes de software, Geração automática de dados de testes, Algoritmos evolutivos, Teste de caminhos

1. Introduction

Software testing is a critical element of software quality assurance and represents the final review of specification, design, and coding [Pressman 1997]. It is also one element of a broader topic often referred as Verification & Validation (V&V), which is a set of activities that ensure that the software correctly implements a specific function and it is traceable to customer requirements. Although V&V encompass a wide range of activities, software testing provides the last bastion from which quality can be assessed and faults

*Supported by Brazilian CNPq Council through Research Grant 134107/2004-7.

can be uncovered [Pressman 1997, Deason et al. 1991], even though being an expensive phase of software development cycle in terms of effort and cost [Whittaker 2000]. Ideally, software testing guarantees the absence of faults in the software, but in reality it only reveals the presence of software faults but never guarantees their absence [Myers 1979].

There are two main testing techniques: white box and black box. The former considers the internal structure of the software when generating test data while the latter generates test data for software from its specification [Beizer 1990]. Automating test data generation is an important step in reducing the cost of software development and maintenance [Mansour and Salame 2004, Pargas et al. 1999, Sthamer 1996] and when automating, it is important to look at two aspects: the test data generation tool and test adequacy criterion.

A test data generation tool is responsible for creating test data, while a test adequacy criterion assures the quality of test cases generated and gives information about the end of testing process. Many test data generators have been developed [Mansour and Salame 2004, Berndt et al. 2003, Pargas et al. 1999, Sthamer 1996, Korel 1990], each one using different kinds or variations of existing testing techniques. Test adequacy criterion usually involves coverage analysis, which can be measured based on different aspects of software like statements, branches, all-uses and paths, for instance [Beizer 1995].

Path testing searches the program domain for suitable test cases that covers every possible path in the software under test (SUT) [Sthamer 1996]. However, it is generally impossible to achieve this goal, for several reasons. First, a program may contain an infinite number of paths when the program has loops [Lin and Yeh 2001, Pargas et al. 1999, Sthamer 1996]. Second, the number of paths in a program is exponential to the number of branches in it [Mansour and Salame 2004, Binder 2000] and many of them may be unfeasible[1]. Third, the number of test cases is too large, since each path can be covered by several test cases. For these reasons, the problem of path testing can become a NP-complete problem[2] [Mansour and Salame 2004], making the covering of all possible path computationally impractical.

Since it is impossible to cover all paths in a software, the problem of path testing selects a subset of paths to execute and find test data to cover it. Various test data generation methods have been proposed in the literature; they can be classified in [Hajnal and Forgács 1998]: random test data generators, symbolic evaluators and function minimization methods. Random test data generators select test data randomly from the input variables domain. The symbolic execution methods are static, in the sense that they analyse a program to obtain a set of symbolic representations of each condition predicate along a selected path. The expressions are obtained by attributing symbolic values to the input variables. If the predicates are linear, then the solution can be obtained by using linear programming [McMinn 2004]. Function minimization methods, on the other hand, are dynamic, since they are based on program execution. They perform an exploratory search [Korel 1990], in which the selected input variables are modified by an small amount and submitted to the program.

[1] A path is *unfeasible* if there is no input for which the path is covered during program execution.

[2] The class of all problems that can be solved in polynomial time on a non-deterministic computer, but merely exponential or worse time on a deterministic computer [NIST 2000].

In such cases the use of metaheuristics[3] would be very useful in providing usable results in a reasonable time. Several metaheuristics have been suggested for path coverage [Mansour and Salame 2004, Lin and Yeh 2001], besides statement [Pargas et al. 1999] and branch [Pargas et al. 1999, Sthamer 1996] coverage. One of these methods that have been used recently for this kind of problem is the Genetic Algorithm (GA).

The GA is a global search metaheuristic proposed originally by Holland [Holland 1975]. Extensive work has been done on the development of the original algorithm in the last 20 years and it has been applied successfully in many fields of science and engineering [Deb et al. 2004a, Deb et al. 2004b]. The GA is based on the principles of Darwin´s theory of natural evolution and belongs to a more general category, the Evolutionary Algorithms (EAs).

Recently, a new evolutionary algorithm was proposed. Called Generalized Extremal Optimization (GEO) [Sousa et al. 2004a, Sousa et al. 2003, Sousa 2002], it 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]. GEO was devised to be easily applicable to a broad class of nonlinear constrained optimization problems, with the presence of any combination of continuos, discrete and integer variables. It has been applied successfully to real optimal design problems [Galski et al. 2004, Sousa et al. 2004b, Sousa et al. 2003] and shown to be competitive to other stochastic methods in benchmark test functions [Sousa et al. 2003, Sousa 2002]. Having only one free parameter to adjust, it can be easily set to give its best performance for a given application. This is an *a priori* advantage over methods such as the GAs since they have at least three parameters to be set, making their tuning to a particular application more prone to be computationally expensive and becoming a problem in itself.

In this work an initial assessment of the efficacy of GEO as a test data generation tool in a pathwise test data generator for path testing is made. A pathwise test data generator is a system that tests software using a testing adequacy criterion [Sthamer 1996]. It consists of a program control flow graph construction, path selection and test data generation tool. The SUT control flow graph is constructed with its edges labeled following program instrumentation on each SUT branch; a list of SUT's paths to be covered is extracted from this graph, and GEO guides its search for test data to cover a specific path creating new test data from previously ones that were evaluated as good candidates. The path test problem is transformed in an optimization problem and each test data candidate is evaluated by a fitness function that reports quantitatively how far the path covered by this candidate is from the path that needs to be covered. The results obtained with GEO are compared to results from a Simple Genetic Algorithm (SGA) and with a random search, called Random-Test.

In the next Section the concept of EAs is introduced, the application of GAs in software testing is briefed and the GA used in this paper for performance comparison with GEO described. In Section 3. GEO is presented, and a discussion on the motivation of applying it to software testing is made. In Section 4. the benchmark test problem is introduced, together with a description of how it was transformed into an optimization problem. In Section 5. results are presented and discussed, followed by the conclusions

[3] A high-level strategy that guides other heuristics in a search for feasible solutions [NIST 2000].

and an outline of future work in Section 6..

2. Evolutionary Algorithms and the Simple Genetic Algorithm

Although the idea of using the principles of natural evolution on problem solving dates back to the late 1940s, it was during the 1960s that the first evolutionary algorithms, the Evolution Strategies (ES) and the Evolutionary Programming (EP), emerged [Eiben and Smith 2003]. Together with the GA and the Genetic Programming (GP) [Koza 1992], they make up the four main branches of the EAs.

The main idea behind the EAs is to evolve a population of individuals (candidate solutions for the problem) through competition, mating and mutation, so that the average quality of the population is systematically increased in the direction of the solution of the problem at hand. The evolutionary process of the candidate solutions is stochastic and "guided" by the setting of adjustable parameters [Eiben and Smith 2003]. In an analogy with a natural ecosystem, in a EA different organisms (solutions) coexist and compete. The more adapted to the design space will be more prone to reproduce and generate descendants. On the other hand, the worst individuals will have fewer or no offspring. In an optimization problem, the fitness of each individual is proportional to the value of the objective (cost) function, also called fitness function.

GAs are probably the most well known and used form of EAs. They have been used effectively in generating test data to reach a specific coverage criterion. For instance, Pargas et al. [Pargas et al. 1999] uses a GA to generate test data candidates that reaches statement and branch coverage; Mansour and Salame [Mansour and Salame 2004] and Lin and Yeh [Lin and Yeh 2001] applies it in test data generation for path testing; Sthamer [Sthamer 1996] does a rich study about GAs and its many internal process variations in his thesis. Moreover, Godefroid and Khurshid [Godefroid and Khurshid 2002] uses GAs to guide a state-space search toward error states of concurrent reactive systems.

In this work the Simple Genetic Algorithm (SGA), as described by Goldberg [Goldberg 1989], was used for comparison with GEO. Although there are many GA implementations with different strategies [Mansour and Salame 2004, Lin and Yeh 2001, Pargas et al. 1999, Sthamer 1996], the SGA was chosen since it is the standard of its family, making it very suitable for the purposes of this initial study. A high level description of the SGA can be seen in Figure 1 below.

```
initialize population P
evaluate fitness on each individual in P
repeat
    select parents from P according to selection mechanism
    recombine parents to form new offspring P' using crossover and mutation
    evaluate fitness of each individual in P'
    P ⟵ P'
until stopping condition reached
```

Figure 1. Pseudo-code for a SGA

In the SGA, each individual of the population, also called chromosome, is represented by a binary string, which encodes the design variables. The population is initialized

randomly attributing the value '0' or '1' to each bit of each individual. The SGA uses fitness proportional selection, one-point crossover and simple bit-flip mutation in its internal processes. These major components are described below:

- *Selection*: The selection of parents for reproduction is done according to a probability distribution based on the individuals fitness values. A "roulette wheel" with slots sized according to fitness is used. First, calculate the fitness value fit_i for each chromosome *i* of the population, where $1 \leq i \leq popsize$ (population size). Then, find the total fitness of the population *TFit*, using equation:

$$TFit = \sum_{i=1}^{popsize} fit_i \tag{1}$$

  Calculate the probability of selection p_i for each chromosome *i*, so that $p_i = \frac{fit_i}{TFit}$. The next step calculates a cumulative probability k_i for each chromosome *i* with equation:

$$k_i = \sum_{j=1}^{i} p_i \tag{2}$$

  The selection process is based on spinning the roullete wheel *popsize* times, and each time a parent is selected as follows: generate a random real number *r* in the range [0,1]; then select chromosome *i* such that

$$i = \begin{cases} \text{first chromosome} & \text{if } r < k_1, \\ \textit{i-th} \text{ chromosome} & \text{where } 2 \leq i \leq popsize \text{ such that } k_{i-1} < r \leq k_i. \end{cases} \tag{3}$$

  Note that some chromosomes may be selected more than once.
- *Reproduction (Crossover)*: In one-point (or single) crossover, two chromosomes selected as potencial parents by selection process exchange substring information at a random position in the chromosome to produce two new chromosomes.

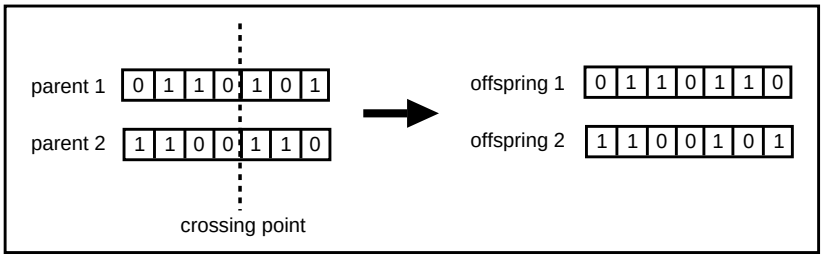

Figure 2. One-point (or single) crossover example

Crossover happens according to a crossover probability p_c, which is an adjustable parameter. For each parent selected, generate a random real number *r* in the range [0,1]; if $r < p_c$ then select the parent for crossover. After that, the selected chromosomes are mated randomly. Then generate a random integer number *pos* in the range [1,m-1], where *m* is the number of bits in a chromosome. This number *pos* indicates the crossing point. Each pair of parents generates two new chromosomes, called offspring.

- *Mutation*: Mutation is performed on a bit-by-bit basis. Every bit of every chromosome in the offspring has an equal chance to mutate (change from '0' to '1' or from '1' to '0'), and the mutation occurs according to a mutation probability p_m, which is also an adjustable parameter. To perform mutation, for each chromosome in the offspring and for each bit within the chromosome, generate a random real number *r* in the range [0,1]; if $r < p_m$ then mutate the bit.

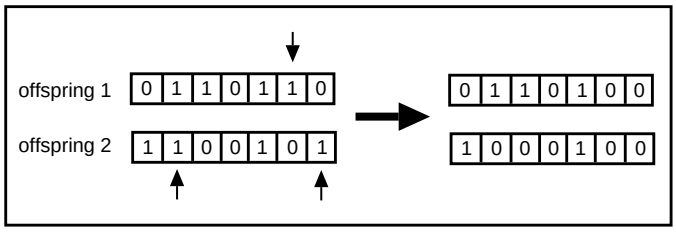

Figure 3. Normal mutation example

These major components including the fitness function will evolve chromosomes to better ones, trying to find a candidate that covers the target path. The crossover process tries to create better individuals from fitter ones, while mutation introduces diversity into population, avoiding getting stucked at local optima solutions.

In the following Section the GEO algorithm is described.

3. The Generalized Extremal Optimization Algorithm

The Generalized Extremal Optimization Algorithm (GEO) is a newly proposed metaheuristic suitable to tackle complex optimization problems. It was developed as a generalization of the Extremal Optimization (EO) method in such a way that it can be readily applied to a broad class of problems. Either EO or GEO are based on the simplified evolutionary model of Bak-Sneppen [Bak and Sneppen 1993], which was devised to show the emergence of Self-Organized Criticality (SOC) in ecosystems. The theory of SOC has been used to explain the power law signatures that emerge from many complex systems in such different areas as geology, economy and biology [Bak 1996]. It states that large interactive systems evolve naturally to a critical state where a single change in one of its elements generates "avalanches" that can reach any number of elements in the system. The probability distribution of the sizes s of these avalanches is described by a power law in the form,

$$P(s) \approx s^{-\tau} \tag{4}$$

where τ is a positive parameter. That is, smaller avalanches are more likely to occur than big ones, but even avalanches as big as the whole system may occur with a non-negligible probability. This kind of dynamic behavior, according to Bak and Sneppen [Bak and Sneppen 1993], could explain the bursts of evolutionary activity observed in the fossil record and that has been given the name of punctuated equilibrium [Gould and Eldredge 1993].

In the Bak-Sneppen model, species are represented in a lattice and, for each of them, is associated a fitness number in the range [0,1]. The evolution is simulated forcing the least adapted species, the one with the least fitness, and their neighbors, to change (it can "evolve" or be "extinct" and replaced by a new one, that has not necessarily a better

fitness). This is done by assigning new fitness numbers, randomly, to these species. This simulated ecosystem is started with the fitness of the species distributed uniformly in the range [0,1]. Since the less adapted species are constantly forced to change, the average fitness value of the ecosystem increases and, eventually, some time after initialization all species have a fitness number above a "critical level". However, as even good species may be forced to change (if they are neighbors of the least adapted one), it happens that a number of species may fall below the critical level from time to time. That is, the equilibrium (being above the critical level in "stasis") of one or more species is punctuated by avalanches, whose occurrence is described by a power law [Bak and Sneppen 1993]. An optimization heuristic based on the Bak-Sneppen model may evolve solutions quickly, systematically mutating the worst individuals, preserving at the same time throughout the search process the possibility of probing different regions of the design space, which is done via avalanches.

Based on the Bak-Sneppen model, Boettcher and Percus developed the EO method to tackle hard problems in combinatorial optimization. In their method, differently from the GA, the fitness value is associated with the design variable [Boettcher and Percus 2001]. However, as they pointed out, in some cases this may become an ambiguous or even impossible task. The GEO algorithm was devised to overcome this problem, so that it could be easily applicable to a broad class of problems, with any kind of design variable, either continuous, discrete or a combination of them, in a design space that may be multimodal or even discontinuous and subject to any kind of constraint.

In GEO each bit is a species, and a string of L bits is considered a population of species. The string encodes the M design variables. For each of them is associated a fitness number that is proportional to the gain (or loss) the objective function value has in mutating (flipping) the bit. Then all bits are ranked from 1, for the least adapted bit, to L for the best adapted and after this, a bit is mutated according to the probability distribution $P \propto k^{-\tau}$, where k is the rank of a selected bit candidate to mutate, and τ a free control parameter. Making $\tau \rightarrow 0$ the algorithm becomes a random search, while for $\tau \rightarrow \infty$, we have a deterministic search. It has been observed that the best value of τ (the one that yields the best performance of the algorithm) for a given application generally lies within a small range, usually between [1,10]. This, added to GEO having only one free parameter, makes the algorithm easily settable to give its best performance on the problem that is being tackled. After the bit is mutated, the procedure is repeated until a given stopping criterion is reached, and the best configuration of bits (the one that gives the best value for the objective function) found is returned.

In a variation of the canonical GEO described above, the bits are ranked separately for each substring that encodes each design variable, and N bits, one for each variable, are flipped at each iteration of the algorithm. The flowchart in Figure 4 shows the main characteristics of GEO and its variation, GEO$_{var}$.

The number of bits needed to represent each design variable depends on the their range and precision. For continuous variables, the minimum number *m* of bits necessary

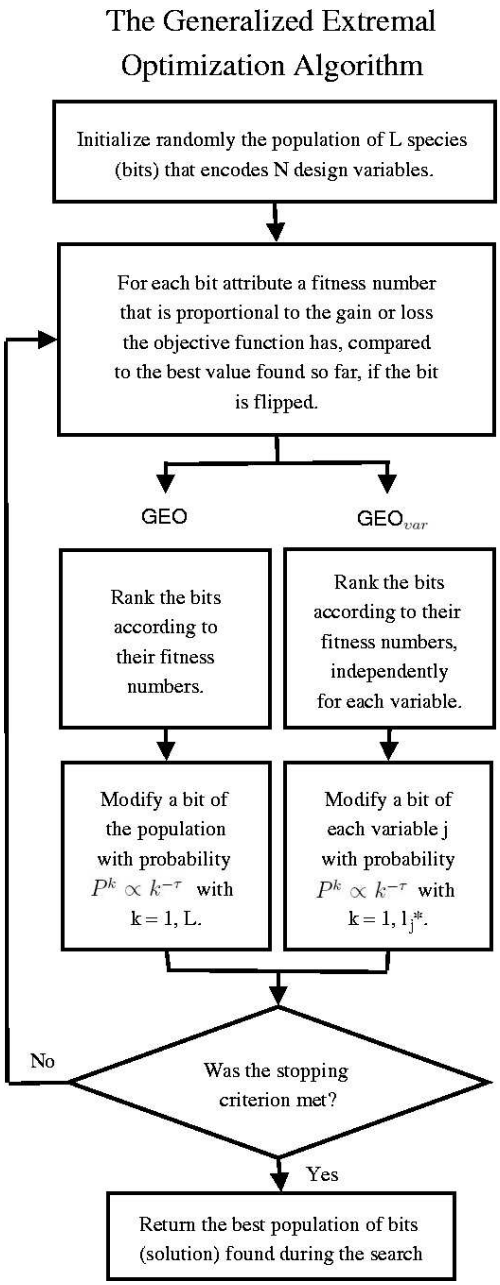

Figure 4. Flowchart of GEO and GEO$_{var}$, where *l_j is the number of bits of each variable *j*, with $j = 1, N$ (from [Sousa et al. 2003])

to achieve a certain precision is given by equation

$$2^m \geq \left[\frac{\left(x_j^u - x_j^l\right)}{p} + 1 \right] \tag{5}$$

where x_j^l and x_j^u are the lower and upper bounds, respectively, of the variable *j*, where $j = 1, N$, and *p* is the desired precision. The real value of each design variable is obtained with equation

$$x_j = x_j^l + \left(x_j^u - x_j^l\right) \cdot \left[\frac{I_j}{(2^m - 1)} \right] \tag{6}$$

where I_j is the integer number obtained in the transformation of variable *j* from its binary form to a decimal representation and *m* the number of bits necessary to represent *j*.

In the next Section the case study used to assess the performance of GEO, the triangle classifier problem, is described.

4. The triangle classifier problem

The Triangle Classifier Program is a benchmark for many researchers in software testing. It was used by Glenford Myers in his book [Myers 1979] and had been applied to compare different heuristics for test case generation by Michael et al. [Michael et al. 2001], Wegener et al. [Wegener et al. 2001], Lin and Yeh [Lin and Yeh 2001], Pargas et al. [Pargas et al. 1999] and Borgelt [Borgelt 1999], as seen on [Berndt et al. 2003]. Its purpose is to classify a triangle as invalid, scalene, isosceles or equilateral, given the length of the three triangle sides. It is important to mention that this program is used so regurlarly because even when using a large range of integers, only a few combinations will satisfy particular branches in code. The program code can be seen in Listing 1.

Listing 1. Instrumented triangle program code, taken from [Lin and Yeh 2001] and adapted to Java programming language

```
public int classifyTriangle(int a, int b, int c) {
  int triangle = 0;                                    // B1
  if((a + b > c) && (b + c > a) && (c + a > b)) {      // B1
    concatPath("u");                                   // Instr.
    if((a != b) && (b != c) && (c != a)) {             // B2
      concatPath("y");                                 // Instr.
      // Scalene triangle
      triangle = 1;                                    // B3
    }
    else {                                             // B4
      concatPath("v");                                 // Instr.
      if(((a == b) && (b != c)) || ((b == c) && (c != a))
            || ((c == a) && (a != b))) {               // B4
        concatPath("z");                               // Instr.
        // Isosceles triangle
        triangle = 2;                                  // B5
      }
      else {                                           // B6
        concatPath("x");                               // Instr.
        // Equilateral triangle
        triangle = 3;                                  // B6
      }
    }
  }
  else {                                               // B7
    concatPath("w");                                   // Instr.
    // Not a triangle
  }
  return triangle;                                     // B8
}
```

The static structure of a program or procedure can be represented by a control flow graph (CFG) [Pressman 1997]. A CFG is a representation of a program where contiguous regions of code without branches, known as basic blocks, are represented as nodes in a graph and edges between nodes indicate the possible flow of the program. A cycle in a CFG may imply that there is a loop in the code. In most presentations there are two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. The CFG for the triangle program, with four paths (*w*, *uy*, *uvz* and *uvx*) is shown in Figure 5.

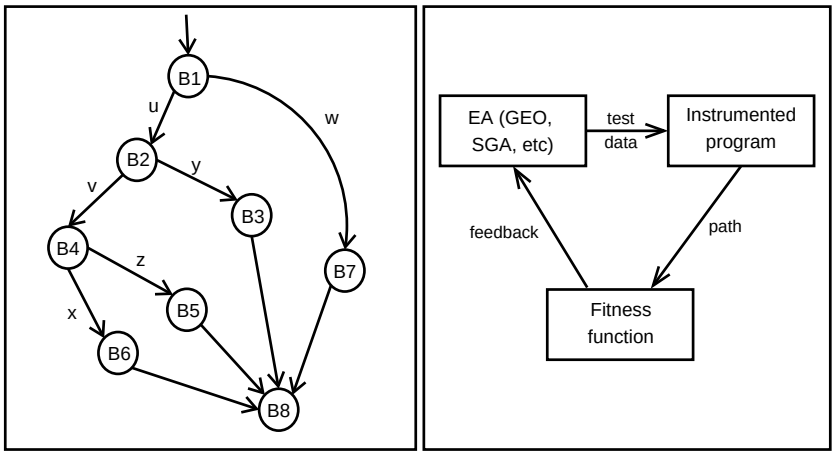

Figure 5. At the left, the CFG of the triangle program, where $B1 - B8$ indicate the basic blocks and $B1$ and $B8$ are the entry and exit block, respectively. At the right, the core process adopted for test data generation using an EA.

The program code was manually instrumented so it would be able to know the path covered by each test data candidate generated. Note that each graph edge (that represents a branch) was labeled following the instrumentation inserted in program code. The most difficult path for random search is path *uvx*, because it has to find three equal integer values in a range of [0, 2^{16}-1] values. In order to evaluate a candidate solution, each design variable value has to be converted to an integer value and will be input to the triangle program. This program outputs the path that was covered with the test data candidate, which will be input to the fitness function, which gives feedback to the EA about the generated test data candidate. The fitness function is described below on next Subsection.

4.1. Fitness function

The fitness function chosen to evaluate each test data candidate was *Similarity*, proposed by Lin and Yeh [Lin and Yeh 2001]. This fitness function extends the Hamming Distance[4] and has the ability to quantify the distance between two given paths.

Given a target path and a current one, the similarity between them is calculated from *n*-order sets of ordered and cascaded branches for each of the paths being compared. After that, for each order, the distance between the set of each path is given by the symmetric difference between them[5]. Then, this distance is normalized to become a real number and the similarity between these *n*-order sets is found by subtracting the value 1 from this normalized distance. Finally, the total similarity between two paths will be the sum of the similarities of all *n*-order sets, each one associated with a weighting factor which is usually found by experience. For further details, take a look at the work of Lin and Yeh [Lin and Yeh 2001].

Similarity was chosen for some reasons. First, it is a fitness function specially developed for path testing. Second, it does not take into account if test data values are closer to boundaries; even though boundary test data are more likely to reveal faults

[4] According to the NIST [NIST 2000], the Hamming Distance is the number of bits which differ between two binary strings.

[5] The symmetric difference between set α and β is a set containing the elements either in α or β but not in both ($\alpha \oplus \beta$).

[Clarke 1976], this is not the focus of this work. Third, its implementation is very easy. Although no real experiment was made with a program with loops, the authors says that their fitness function deals well with program paths with loops.

In Section 5. below the results are shown and discussed.

5. Results

The comparison among GEO, SGA and Random-Test algorithms was made based on the number of times each of them found the target path in a predefined set of runs. All algorithms were implemented in Java, so that it would be easier to integrate GEO in Eclipse Integrated Development Environment (IDE) on future research developments of this work.

The Random-Test approach uses random search and selects arbitrarily test data from the input domain and then these test data are applied to the SUT. Test data generation is driven by attributing randomly the value '0' or '1' to every bit needed to represent a test data candidate. Then this candidate is converted to an integer value and is ready to be applied to the SUT.

Because the performance of either GEO or SGA may vary significantly with the values of the adjustable parameters, a series of experiments were made with each of the algorithms to get them properly tuned to the path testing problem being tackled. GEO has only one adjustable parameter, τ, while SGA has three: mutation (p_m) and crossover (p_c) probabilities, and population size (*popsize*). The tuning experiments conditions were:

- Each experiment is equivalent to a set of 100 runs with a parameter configuration.
- The maximum number of evaluations for *Similarity*, at each run, were 100000.
- The most difficult path, *uvx*, was used as target.
- A successful run occurs when the input test data generated covers the target path.
- GEO and GEO$_{var}$ started from the same randomly generated points.
- For GEO and GEO$_{var}$: $0.75 \leq \tau \leq 10$, starting at 0.75, with increments of 0.25 for each experiment.
- For SGA:

$$\begin{cases} 100 \leq popsize \leq 10000 & \text{starting at 100 and with increments of } \textit{popzise} * 10, \\ 0.6 \leq p_c \leq 1.0 & \text{starting at 0.6 and with increments of 0.1}, \\ 0.0010 \leq p_m \leq 0.0205 & \text{starting at 0.001 and with increments of 0.0015} \end{cases}$$

It's easy to see that GEO's tuning is easier than SGA. The former has 38 different parameter configurations, while the latter has 210 ($3 * 5 * 14$). For the SGA experiments, the number of generations depends on $popsize$, since the total number of function evaluations at each run (100000) must be the same for all algorithms. Note that Random-Test algorithm doesn't require tuning.

All the experiments were made using the type **integer**, which is represented by 16 bits, and only positive integers were considered. For the triangle program, there are three design variables, each one in the range [0,65535]. The GEO τ increment of 0.25 was chosed based on previous experiments with it, while the SGA ones were chosed in order to evaluate in a more refined way its performance regarding the parameters values.

The graph in Figure 6 shows the influence of τ in the performance of GEO. It is clear from Figure 6 that GEO$_{var}$ performed better than GEO for this problem. The best

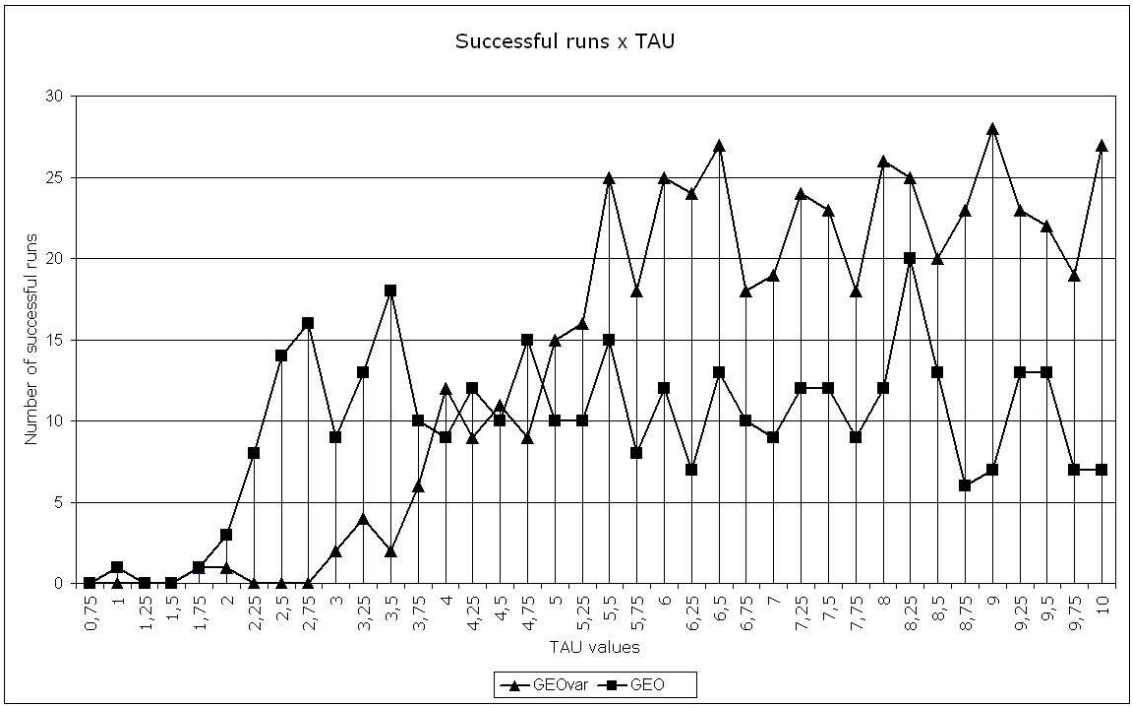

Figure 6. Tuning GEO and GEO$_{var}$ free parameter τ

τ value for GEO was 3.5 and for GEO$_{var}$, 9.0, which makes also clear that better performances are achieved with higher values of τ. That is, for this problem the performance of GEO is increased as it is tuned to be more deterministic, what makes the least adapted bit of the string be almost always mutated.

The tuning parameters and results for the SGA are shown in Table 1. The best results were obtained with a $popsize$ of 100 individuals, p_c of 1.0 and p_m of 0.016, which yielded 31 successful runs.

Table 1. Tuning SGA's population size, crossover and mutation probabilities

	$popsize$ of 100					$popsize$ size of 1000					$popsize$ of 10000				
	Crossover probability					Crossover probability					Crossover probability				
Mutation probability	0.6	0.7	0.8	0.9	1.0	0.6	0.7	0.8	0.9	1.0	0.6	0.7	0.8	0.9	1.0
0.0010	0	0	0	0	0	15	17	12	19	12	5	4	6	0	5
0.0025	3	1	4	1	3	15	18	16	17	14	3	**8**[a]	7	3	2
0.0040	5	4	6	6	8	21	19	17	17	16	3	2	6	3	0
0.0055	7	9	6	6	8	14	12	22	19	17	2	6	2	2	1
0.0070	8	13	13	10	15	20	22	19	22	14	2	6	5	5	5
0.0085	10	14	15	15	14	14	13	21	18	18	1	3	3	0	0
0.0100	14	12	9	14	17	19	19	25	26	21	1	3	1	1	4
0.0115	18	16	12	17	19	17	**30**[b]	20	17	17	3	0	1	1	1
0.0130	16	18	18	19	20	23	21	26	20	20	1	1	0	0	0
0.0145	19	20	15	22	22	26	22	19	15	17	5	2	1	0	0
0.0160	19	17	14	18	**31**[c]	21	19	22	26	18	2	1	2	1	1
0.0175	22	13	14	22	24	19	20	17	23	22	0	0	1	1	0
0.0190	13	22	14	23	19	14	20	21	23	22	0	0	0	1	0
0.0205	17	16	17	21	23	15	20	23	19	16	0	0	0	0	0

[a] Best number of successful runs for $popsize$ of 10000

[b] Best number of successful runs for $popsize$ of 1000

[c] Best number of successful runs for $popsize$ of 100

Although the tuning was made using the most difficult path as the target path, the practical process usually involves choosing an easier one, because it is less time consuming. Then the best parameter configuration found can be applied to a more difficult path. However, it must be noted that for a new problem it may be necessary reset the free parameters.

After the tuning process, the best parameter configurations were set in the algorithms trying to find test data to cover each one of the four paths of the triangle program. The number of experiments made for each algorithm with its best parameter configuration was 20. As in the tuning process, each experiment had 100 runs and the algorithms were compared using the average number of successful runs for all the 20 experiments. The results are shown in Table 2.

Table 2. Average number of successful runs for each algorithm

Target path	GEO	GEO$_{var}$	SGA	Random-Test
uvx	11	23	19	0
uvz	64	72	99	96
uy	100	100	100	100
w	100	100	100	100

GEO$_{var}$ clearly outperforms GEO and also the SGA when the target path was the most difficult one. However, GEO had some difficulty in finding test data when the target path was *uvz*, which can be seen by the lower number of successful runs when compared with SGA and Random-Test algorithms.

As we look to the design space for the triangle program, it's easy to see that just a few points at this space yields an equilateral triangle. Also, there are more points yielding an escalene than the isosceles one. GEO starts its search from a unique point at the design space, while SGA starts in *popsize* different points, increasing its chance to find test data for the path *uvz* in the beginning of the search earlier than GEO. However, the experiments showed that even with this disadvantage, GEO$_{var}$ outperformed SGA while searching for data to cover the most difficult path.

Another interesting result was the comparison made between the evolution of GEO and SGA's best fitness values. GEO has L bits solutions at each iteration, while the SGA has $popsize$ solutions in each generation. One experiment of those 20 was selected. For each run of these experiments, starting from 0 to 100000 *Similarity* evaluations and with increments of 1000, GEO and SGA´s best fitness values were collected. Then, an average of the 100 runs was made and results are shown in Figure 7.

The SGA average best fitness value increased quickly during the first 32000 *Similarity* evaluations and reached the value of 1.5, due to the fact that almost all the runs at that point had already found test data yielding the fitness value of 1.5, which is test data that covers path *uvz*. After that, path *uvx* test data started to be found and contributed to raise the SGA average fitness value until 3.03, with 100000 *Similarity* evaluations. Note in the SGA curve that most of the successful runs for it happened in the range of 32000 to 100000 *Similarity* evaluations. On the other hand, although GEO$_{var}$ average best fitness value evolved slowly at the first moment, it had a better number of successful runs. Most of them also happened in the same range as the SGA, but GEO$_{var}$'s increase rate was

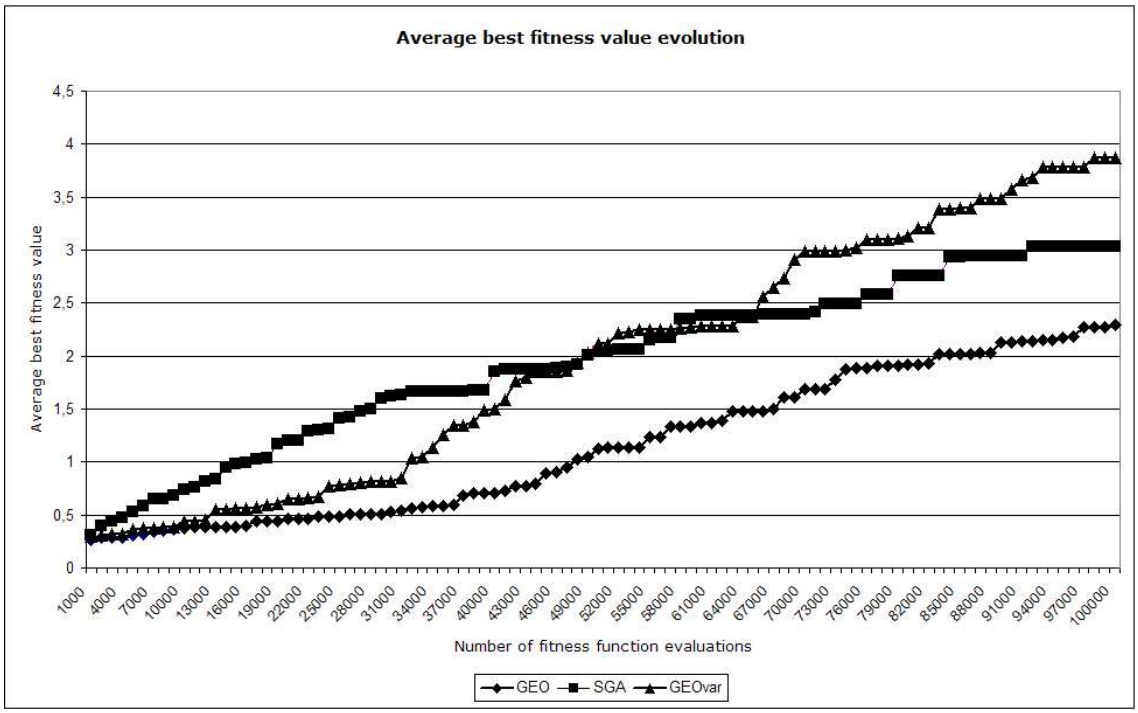

Figure 7. Evolution of GEO and SGA´s average best fitness value

clearly superior than SGA's, as showed by its curve. This higher rate explains the fact that GEO$_{var}$ performed better than SGA while finding test data for path *uvx*. The Figure also shows that SGA and GEO$_{var}$ outperformed GEO, confirming data from Table 2.

6. Conclusions

The results showed that the GEO algorithm is very competitive with the SGA for this initial assessment. GEO's variation, GEO$_{var}$, outperformed SGA while searching for test data to cover the most difficult path in the triangle problem, proving its efficacy. GEO had never been used to tackle Software Engineering problems, and its performance as a test data generation tool for path testing was very good. Its tuning process was extremely simple, while SGA's was too much costly.

In future developments of this work it is intended to increase the size of the problem, applying the same approach for programs with more paths and also including loops. Moreover, other kinds of fitness functions would be explored, in order to verify its influence on the performance of GEO. For example, a fitness function that takes into account the boundaries of the predicates in a path. In future performance evaluations, more sophisticated versions of the GA would be used. Comparison with other popular metaheuristic, the Simulated Annealing (SA), is also envisioned.

References

Bak, P. (1996). *How Nature Works: The Science of Self-Organized Criticality*. Springer.

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.

Beizer, B. (1995). *Black-Box Testing*. John Wiley & Sons.

Berndt, D., Fisher, J., Johnson, L., Pinglikar, J., and Watkins, A. (2003). Breeding Software Test Cases with Genetic Algorithms. In *36th Annual Hawaii Int. Conference on System Sciences (HICSS'03)*.

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

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

Borgelt, K. (1999). Software Test Data Generation from a Genetic Algorithm. In Karr, C. L. and Freeman, L. M., editors, *Industrial Applications of Genetic Algorithms*, pages 49–68. CRC Press, Boca Raton, FL.

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

Deason, W. H., Brown, D. B., Chang, K. H., and II, J. H. C. (1991). A Rule-Based Software Test Data Generator. *IEEE Trans. on Knowl. and Data Eng.*, 3(1):108–117.

Deb, K., Poli, R., Banzhaf, W., Beyer, H., Burke, E. K., Darwen, P. J., Dasgupta, D., Floreano, D., Foster, J. A., Harman, M., Holland, O., Lanzi, P. L., Spector, L., Tettamanzi, A., Thierens, D., and Tyrrell, A. M., editors (2004a). *Genetic and Evolutionary Computation Conference - GECCO, Seattle, WA, USA, 2004, Proceedings, Part I*, volume 3102 of *Lecture Notes in Computer Science*. Springer.

Deb, K., Poli, R., Banzhaf, W., Beyer, H., Burke, E. K., Darwen, P. J., Dasgupta, D., Floreano, D., Foster, J. A., Harman, M., Holland, O., Lanzi, P. L., Spector, L., Tettamanzi, A., Thierens, D., and Tyrrell, A. M., editors (2004b). *Genetic and Evolutionary Computation Conference - GECCO, Seattle, WA, USA, June 26-30, 2004, Proceedings, Part II*, volume 3103 of *Lecture Notes in Computer Science*. Springer.

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)*.

Godefroid, P. and Khurshid, S. (2002). Exploring Very Large State Spaces Using Genetic Algorithms. In *TACAS '02: Proc. of the 8th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280. Springer.

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

Gould, S. J. and Eldredge, N. (1993). Punctuated equilibrium comes of age. *Nature*, 366:223–227.

Hajnal, A. and Forgács, I. (1998). An applicable test data generation algorithm for domain errors. In *ISSTA '98: Proc. of the 1998 ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, pages 63–72, New York, NY, USA. ACM Press.

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

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

Koza, J. R. (1992). *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press.

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.

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. Software Eng.*, 27(12):1085–1110.

Myers, G. J. (1979). *Art of Software Testing*. John Wiley & Sons.

NIST (2000). National Institute of Standards and Technology. `http://www.nist.gov`. Last access on 03/13/2005.

Pargas, R. P., Harrold, M. J., and Peck, R. (1999). Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282.

Pressman, R. S. (1997). *Software Engineering: A practitioner's approach*. McGraw-Hill, 4th edition.

Sousa, F. L. (2002). *Otimização Extrema Generalizada: Um novo algoritmo estocástico para o projeto ótimo*. PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE). INPE-9564-TDI/836.

Sousa, F. L., Ramos, F. M., Galski, R. L., and Muraoka, I. (2004a). Generalized Extremal Optimization: A New Meta-heuristic Inspired by a Model of Natural Evolution. In L. N. de Castro, F. J. V. Z., editor, *Recent Developments in Biologically Inspired Computing*. Idea Group Inc.

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. (2004b). 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.

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

Whittaker, J. A. (2000). What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–79.