Software Testing with Evolutionary Strategies

Enrique Alba and J. Francisco Chicano

Departamento de Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN
eat@lcc.uma.es chicano@lcc.uma.es

Abstract. This paper applies the Evolutionary Strategy (ES) metaheuristic to the automatic test data generation problem. The problem consists in creating automatically a set of input data to test a program. This is a required step in software development and a time consuming task in all software companies. We describe our proposal and study the influence of some parameters of the algorithm in the results. We use a benchmark of eleven programs that includes fundamental algorithms in computer science. Finally, we compare our ES with a Genetic Algorithm (GA), a well-known algorithm in this domain. The results show that the ES obtains in general better results than the GA for the benchmark used.

1 Introduction

Automatic test data generation consists in proposing automatically a "good" set of input data for a program to be tested. This is a very important, time consuming, and hard task in software development [1]. But, what is a good set of data inputs? Intuitively, we can state that a good set of test data will allow a large amount of faults in a program to be discovered. For a more formal definition we have to resort to the *test adequacy criteria* [2].

The automatic generation of test data for computer programs has been tackled in the literature since many years ago [3, 4]. A great number of paradigms has been applied to the test data generation. In the next paragraphs we mention some of these techniques.

A first paradigm is the so-called *random test data generation*. The test data are created randomly until the test adequacy criterion is satisfied or a maximum number of data sets is generated. We can find some experiments with random test data generators in [5] and more recently in [2].

Symbolic test data generation [3] consists in using symbolic values for the variables instead of real values to get a symbolic execution. Some algebraic constraints are obtained from this symbolic execution and these constraints are used for finding test cases. Godzilla [6] is an automatic test data generator that uses this technique.

A third and widely spread paradigm is *dynamic test data generation*. In this case, the program is instrumented to pass information to the test generator. The test generator checks whether the test adequacy criterion is fulfilled or not. If the criterion is not fulfilled it prepares new test data to serve as input for the

program. The test data generation process is translated into a function minimization problem, where the function is some kind of "distance" to an execution where the test criterion is fulfilled. This paradigm was presented in [4] and there are many works based on it [1, 2, 7, 8].

Following the dynamic test data generation paradigm, several metaheuristic techniques have been applied to the problem in the literature. Mantere and Alander in [9] present a recent review on the applications of the evolutionary algorithms to software testing. Most of the papers included in their discussion use GAs to find test data. In fact, only a few works listed in the review include other techniques such as cultural algorithms [10] (a special kind of GA), hill climbing [11], and simulated annealing [12].

We have found other recent works applying metaheuristic algorithms to software testing. In [13] the authors explain how a Tabu Search algorithm can be applied to generate test data obtaining maximum branch coverage. Sagarna and Lozano tackle the problem by using an Estimation of Distribution Algorithm (EDA) in [14] and they compare a Scatter Search (SS) with EDAs in [15].

In this work we propose the ES for finding input data sets in software testing. To our knowledge, this is the first work applying ES to this domain. This technique has some advantages such as self-adaptability, and real number representation of the problem variables. The former reduces the human effort in tuning the algorithm. The last makes possible to explore a wide region of the input values.

The rest of the paper is organized as follows. We detail the construction of our test data generator in the next section. Section 3 gives a description of the Evolutionary Strategy. Then, in Sect. 4 we analyze the experiments on the benchmark. Finally, Sect. 5 addresses the final conclusions and future work.

2 The Test Data Generator

In this section we describe the test data generator proposed and the whole test generation process. We are interested in testing the functional requirements of the software, that is, the relationship between input and output. Some other works apply evolutionary techniques in software testing to check non-functional requirements like temporal constraints [16]. As we said in the previous section, in order to formalize the problem we must define a test adequacy criterion. There exist many of them. For example, in the *statement coverage* we require all the statements in the program to be executed. On the other hand, *branch coverage* requires taking all the branches in the conditional statements. We choose in this work the same test adequacy criterion as in [2]: *condition-decision coverage.* To fulfill this criterion all conditions must be true and false at least once after executing all the set of test data on it. A condition is an expression that is evaluated during the program execution to a boolean value (true or false) with no other nested conditions. All the comparison expressions are conditions. On the contrary, a decision is a boolean expression whose value affects the control flow. The boolean expression following an `if` keyword in C is a decision. It

is important to note that full condition-decision coverage implies full branch coverage but not vice versa. That is, if we find a set of test inputs that makes true and false all the program conditions at least once we can ensure that all the decisions will take values true and false and, in consequence, that all branches will be taken; but taking all branches does not ensure that all conditions take the two boolean values. We call *test program* to the program being tested.

Our proposal is a white-box test data generator, so the first step is to instrument the test program itself. To do this, we add some neutral statements to all the conditions in order to inform about the boolean value of the condition and the fitness value needed in the ES (see below). The previous process is done automatically by an application that parses the C source program and generates a modified C source program with the same original behavior. The modified code is compiled and linked with an object file to create the executable file used in the experiments. The process is summarized in Fig. 1.

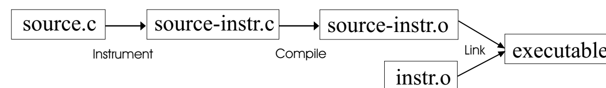

Fig. 1. The instrumentation process.

In the instrumentation process each condition is transformed into an expression that is evaluated to the same value as the original condition. This expression has a side effect: it informs about the condition reached and the value it takes. Our actual version of the instrumentor software is able to transform all non-empty conditions appearing in a decision of a `for` loop, a `while` loop, a `do while` loop, and an `if` statement. At present, we must transform by hand all the `switch` sentences and the `(a?b:c)` expressions into one of the supported kinds.

The object file `instr.o` of Fig. 1 implements some functions that are called by the new added statements to inform about the conditions reached. The final executable file must be executed by using a special program: the *launcher*. This program acts as a bridge between the modified program and the test data generator (see Fig. 2). It writes to the standard output all the information about the conditions executed in the test program.

When the test data generator executes the modified test program with a given input, a report of the condition values is computed and transmitted to the launcher. The launcher writes this report to the standard output and the generator captures it. With this information the generator builds a coverage table where it stores, for each condition, the set of test data that makes the condition true and false along the process. That is, for each condition the table stores two sets: the true set and the false set. This table is an important internal data structure that is looked up during the test generation. We say that a condition is

reached if at least one of the sets associated with the condition is non-empty. On the other hand, we say that a condition is *covered* if the two sets are non-empty.

The outline of the test generator is as follows. First, it generates some random test data (10 in this work) and executes the test program with these data. As a result, some entries of the coverage table will be filled. Then, it looks up in the coverage table for a condition reached but not covered. When found, it uses the ES to search a test data making that condition to take the value not covered yet. During this search the ES can find test data covering other conditions. These test data are also used for updating the condition table. If the ES does not fulfill its objective another reached and not covered condition is chosen and the ES is executed again. The loop stops when all the conditions are covered or when there is no insertion in an empty set of the coverage table during a predefined number of steps (10 steps in this work). Fig. 2 summarizes the test data generator scheme. We have implemented the test data generator in `Java`. In the next section we will detail the evolutionary strategy.

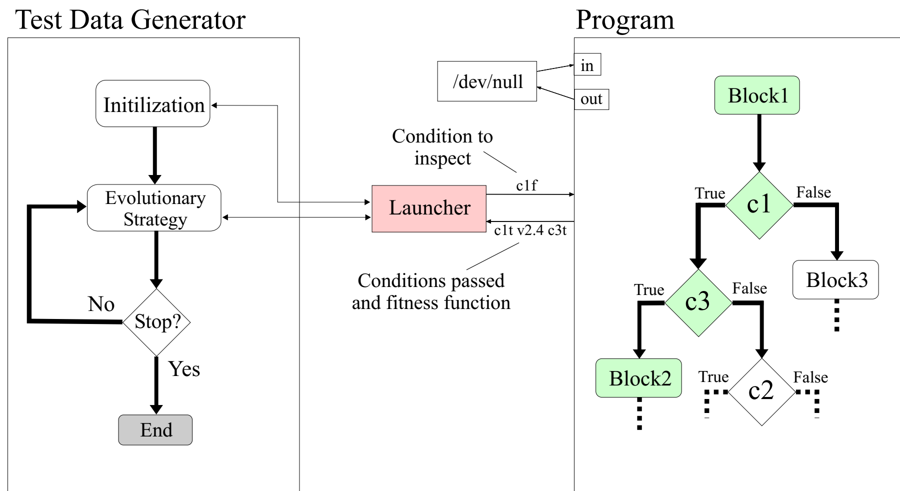

Fig. 2. The test data generation process. The launcher communicates with the test data generator by means of the standard input/output. The communication between the program and the launcher is carried out by using pipes.

3 Evolutionary Strategy

An Evolutionary Strategy is a kind of Evolutionary Algorithm (EA). EAs [17] are heuristic search techniques loosely based on the principles of natural evolution, namely adaptation and survival of the fittest. These techniques have been shown to be very effective in solving hard optimization tasks. They are based

on a set of solutions (individuals) called *population*. The problem knowledge is usually enclosed in a function, the so-called *fitness function*, that assigns a quality value to the individuals. In Fig. 3 we show the pseudocode of a generic EA. Initially, the algorithm creates a population of μ individuals randomly or by using a seeding algorithm. At each step, some individuals of the population are selected, usually according to their fitness values, and used for creating new λ individuals by means of *variation operators*. Some of these operators only affect to one individual (mutation), but others generate a new individual by combining components of several of them (recombination). These last operators are able to put together good solution components that are distributed in the population, while the first one is the source of new different components. The individuals created are evaluated according to the fitness function. To end a step of the algorithm, a replacement operator decides what individuals will form part of the new population. This process is repeated until a stop criterion, such as a maximum number of evaluations, is fulfilled. Depending on the representation of the solutions and the variation operators used we can distinguish four main types of EAs: genetic algorithm, evolutionary strategy, evolutionary programming, and genetic programming.

```
t := 0;
P(0) = Generate ();
Evaluate (P(0));
while not StopCriterion do
   P^1(t) := Select (P(t));
   P^2(t) := VariationOps (P^1(t));
   Evaluate (P^2(t));
   P(t+1) := Replace (P(t),P^2(t));
   t := t+1;
endwhile;
```

Fig. 3. Pseudocode of an Evolutionary Algorithm (EA).

In an Evolutionary Strategy [18] each individual is composed of a vector of real numbers representing the problem variables ($\mathbf{x}$), a vector of standard deviations (σ) and, optionally, a vector of angles (ω). These two last vectors are used as parameters for the main operator of this technique: the Gaussian mutation. They are evolved together with the problem variables themselves, thus allowing the algorithm to self-adapt the search to the landscape. For the recombination operator of an ES there are many alternatives: each of the three real vectors of an individual can be recombined in a different way. However, this operator is not so important as the mutation. In fact, we do not use recombination in our algorithm. The mutation operator is governed by the three following equations:

$$\sigma_i' = \sigma_i \exp(\tau N(0,1) + \eta N_i(0,1)) \ . \tag{1}$$

$$\omega_i' = \omega_i + \varphi N_i(0,1) \ . \tag{2}$$

$$\mathbf{x}' = \mathbf{x} + \mathbf{N}(\mathbf{0}, C(\sigma', \omega')) \ . \tag{3}$$

where $C(\sigma', \omega')$ is the covariance matrix associated to σ' and ω', $N(0,1)$ is the standard univariate normal distribution, and $\mathbf{N}(\mathbf{0}, C)$ is the multivariate normal distribution with mean $\mathbf{0}$ and covariance matrix C. The subindex i in the standard normal distribution indicates that a new random number is generated anew for each component of the vector. The notation $N(0,1)$ is used for indicating that the same random number is used for all the components. The parameters τ, η, and φ are set to $(2n)^{-1/2}$, $(4n)^{-1/4}$, and $5\pi/180$, respectively, as suggested in [19]. With respect to the replacement operator, there is a special notation to indicate wether the old population is taken into account or not to form the new population. When only the new individuals are used, we have a (μ, λ)-ES; otherwise, we have a $(\mu + \lambda)$-ES.

The input values of a test program can only be real or integer numbers. Each of these number values is a problem variable in the ES and, thus, is represented with a real value. This value is rounded in the case of integer arguments when the test program is executed. In this way, the ES can explore the whole solution space. This contrasts with other techniques such as genetic algorithm with binary representation that can only explore a limited region of the space.

The test generator communicates the objective of the search to the modified test program (that is, the condition and its boolean value) by means of the launcher and the program computes the fitness value when it reaches the condition (regardless of the boolean value it takes). The fitness function is designed to be minimized and depends on the kind of target condition and the boolean value it should take. When the objective condition is not reached the fitness function takes a predefined high value (100000). Table 1 summarizes the fitness expressions for each kind of condition and boolean value desired.

Table 1. Fitness expressions for different kinds of conditions and desired boolean values. The variables a and b are numeric variables (integer or real).

Condition type	*true* fitness	*false* fitness
$a < b$	$a - b + 1$	$b - a$
$a <= b$	$a - b$	$b - a + 1$
$a == b$	$(b-a)^2$	$(1 + (b-a)^2)^{-1}$
$a != b$	$(1 + (b-a)^2)^{-1}$	$(b-a)^2$
a	$(1 + a^2)^{-1}$	a^2

The population of the evolutionary strategy is partially seeded with some test data that is known to reach the objective condition. These data are taken randomly from the non-empty set associated to the objective condition in the condition table. The rest of the population is generated randomly.

4 Experiments

We present in this section the experiments performed over a benchmark of eleven test programs in C covering some practical and fundamental computer science aspects. The programs range from numerical computation (such as Bessel functions) to general optimization methods (such as simulated annealing). Most of the source codes have been extracted from the book "C Recipes" available on-line at `http://www.library.cornell.edu/nr/bookcpdf.html`. The programs are listed in Table 2, where we inform about the number of conditions, the lines of code (LOC), the number of input arguments, a brief description of their goal, and the way of accessing them.

Table 2. Programs tested in the experiments. The source column presents the name of the function in C-Recipes.

Program	Conds.	LOC	Args.	Description	Source
`triangle`	21	53	3	Classify triangles	Ref. [2]
`gcd`	5	38	2	Greatest Common Denominator	Authors
`calday`	11	72	3	Calculate the day of the week	`julday`
`crc`	9	82	13	Cyclic Redundant Code	`icrc`
`insertion`	5	47	20	Sort by insertion method	`piksrt`
`shell`	7	58	20	Sort by shell method	`shell`
`quicksort`	18	143	20	Sort by quicksort method	`sort`
`heapsort`	10	72	20	Sort by heapsort method	`hpsort`
`select`	28	200	21	kth element of an unordered list	`selip`
`bessel`	21	245	2	Bessel J_n and Y_n functions	`bessj*,bessy*`
`sa`	30	332	23	Simulated Annealing	`anneal`

The first test program, `triangle`, receives three integer numbers and decides the kind of triangle they represent: equilateral, isosceles, scalene, or no triangle. The next program, `gcd`, computes the greatest common denominator of the two integer arguments. The `calday` test program computes the day of the week given a date as three integer arguments. In `crc` the cyclic redundant code is computed from 13 integer numbers given in the arguments. The next four test programs (`insertion`, `shell`, `quicksort`, `heapsort`) sort the 20 real arguments using well-known sorting algorithms. The `select` program gets the kth element from an unsorted list of real numbers. The first argument is k and the rest of the arguments forms the unsorted list. The next program computes the Bessel functions given an order n and real argument. Finally, the `sa` program solves an instance of the Travelling Salesman Problem with 10 cities by using Simulated Annealing. The three first arguments are seed parameters for the pseudorandom number generator. The rest of the arguments contains real numbers representing the two-dimensional position of each city.

In the following section we present the results with an evolutionary strategy. In Sect. 4.2 we perform a parametric study by changing the number of offsprings

and the population size of the ES. Finally, in Sects. 4.3 and 4.4 we compare our ES with a GA and with previous results in the literature, respectively.

4.1 Results of the Evolutionary Strategy

Our ES has a population size of $\mu = 10$ individuals and the 10% of this population (that is, 1 individual) is seeded by using the mechanism explained in Sect. 2. The number of offsprings is $\lambda = 1$ and the maximum number of iterations of the ES main loop is 100 (110 evaluations). We perform 30 independent runs of the test data generator and we show in Table 3 some coverage measures, the average number of evaluations, and the average time used.

Table 3. Results obtained with the (10+1)-ES for the eleven programs. Average values over 30 independent runs.

Program	**Cov.(%)**	**Corr. Cov.(%)**	**Max. Corr.(%)**	**Std. Dev.**	**Evals.**	**Time(s)**
triangle	97.54	99.92	100.00	0.0044	1975	10.6
gcd	100.00	100.00	100.00	0.0000	21	0.0
calday	91.82	91.82	100.00	0.0246	1182	6.8
crc	94.44	100.00	100.00	0.0000	1114	28.7
insertion	100.00	100.00	100.00	0.0000	10	0.0
shell	100.00	100.00	100.00	0.0000	10	0.0
quicksort	88.89	94.12	94.12	0.0000	1110	8.4
heapsort	90.00	100.00	100.00	0.0000	1110	9.4
select	53.57	83.33	83.33	0.0000	1120	8.9
bessel	95.08	97.40	97.56	0.0061	1306	7.6
sa	97.06	98.70	100.00	0.0072	1329	5082.1

Analyzing the solutions, we found that a 100% of condition-decision coverage is impossible to reach in some test programs because there are conditions that can not be true or false in certain situations. For example, an infinite loop has a condition that is always true (Fig. 4 left). Another example is the condition `(sign(x)>2)`, where the function `sign` can only return three values: -1, 0, +1. Thus, there can be pairs condition-value that are *unreachable*. In the previous cases the test generator can not reach the 100% of coverage due to the test program itself (code-dependent coverage loss), but we can find other reasons that can explain the absence of a perfect coverage: the inaccuracy of the test data generator and the environment in which it executes. One example of the second effect is related to the dynamic memory allocation. Suppose that a program allocates some dynamic memory and then checks if the memory allocation failed. Most probably it succeed and the check condition gets only one value. In this case we have an environment-dependent coverage loss (Fig. 4 right).

The code-dependent coverage loss is always present, no test data generator can get test data to cover the unreachable condition-value pairs. For this reason,

```
while(1)
{
/* The previous condition
is always true */
:
}
```

```
char *a;
p = (char *)malloc (4);
if (!p)
{
        fprintf("Error");
        exit(0);
}
```

Fig. 4. Two pieces of code that prevent from reaching 100% of condition-decision coverage. The left one produces a code-dependent coverage loss, while the right one produces an environment-dependent coverage loss.

we have used another parameter, in addition to the condition-decision coverage, to measure the quality of the test data generator in a more reliable way. This parameter, that we call *corrected coverage*, appears in the third column of Table 3, and it is defined as the ratio between the condition-value pairs reached with the test data and all the potentially reachable condition-value pairs. The difference with the coverage is that the corrected coverage does not take into account the unreachable condition-value pairs. In the rest of this section we use the word coverage for referring to the corrected coverage. The fourth and fifth columns of the table show the maximum and the standard deviation of the corrected coverage for each test program.

Observing the values of Table 3 we can notice a result that justifies the need of the corrected coverage measure. According to the second column, the test data generator gets a higher coverage in the `calday` program than in the `quicksort` program. However, the third column seems to report the opposite: the corrected coverage for `quicksort` is higher than that for `calday`. If we focus on the maximum corrected coverage, the generator reaches a 100% of coverage with `calday` against the 94.12% of `quicksort`. From this result we conclude that the generator is very stable for the `quicksort` program but has an imprecise behavior with the `calday` program (standard deviation above zero). In summary, we can say that our test data generator is better for `quicksort` (more stable and precise) than for `calday`.

Comparing all the test programs with respect to the maximum corrected coverage we can see that only three out of eleven are below 100%. From these "more difficult" programs, `quicksort` and `select` have dynamic memory allocation sentences that never failed in the environment where they are executed.

With respect to the average number of evaluations, we observe that two programs got the very minimum number of them: `insertion` and `shell`. The ten evaluations correspond to the initial random test data generated. In these cases the test data generator gets full coverage with these random data. They are followed by `gcd`, that only needs one iteration of the evolutionary strategy loop to ensure a perfect coverage. This does not mean that it is straight-forward for a human programmer to test them. It just means that they are easy for our algorithm. For the test programs where the corrected coverage does not coincide with

the condition-decision coverage (all test programs but `gcd`, `calday`, `insertion`, and `shell`) we expect a large number of evaluations in the results (confirmed by Table 3 and the tables below). The reason is that the algorithm tries to get 100% of non-corrected coverage, what is impossible due to the unreachable condition-value pairs.

To end this analysis we now focus on the running times. For most of the test programs the evaluation time is very low, as we can deduce comparing the two last columns of Table 3. However, `sa` is very time consuming, followed by `crc`.

4.2 Influence of λ and μ on the results

In a second set of experiments we perform a study on the influence of several parameters of the ES in the results. In particular, we will change the number of offsprings generated at each iteration and the size of the population, that is, the parameters λ and μ. For each program we perform a Student t-test between the different parameterizations of the algorithm. We consider a statistically confidence of 95% (p-value below 0.05). First, we maintain the value of $\mu = 10$ and we try three values of λ: 1, 2, and 3. Table 4 and Fig. 5 summarize the results obtained when varying the number of offsprings λ.

Table 4. Results after changing the number of offsprings λ.

Test	**(10+1)-ES**		**(10+2)-ES**		**(10+3)-ES**	
Programs	Cor.(%)	Evals.	Cor.(%)	Evals.	Cor.(%)	Evals.
triangle	99.92	1975	100.00	3329	100.00	4501
gcd	100.00	21	100.00	21	100.00	21
calday	91.82	1182	95.76	1994	95.91	3004
crc	100.00	1114	100.00	2124	100.00	3120
insertion	100.00	10	100.00	10	100.00	10
shell	100.00	10	100.00	10	100.00	10
quicksort	94.12	1110	94.12	2110	94.12	3110
heapsort	100.00	1110	100.00	2110	100.00	3110
select	83.33	1120	83.33	2119	83.33	3119
bessel	97.40	1306	97.48	2489	97.48	3577
sa	98.70	1329	99.15	2923	99.10	4115

As we can observe in Table 4, the influence of the number of offsprings in the coverage is negligible, only `triangle`, `calday`, `bessel`, and `sa` get a slightly higher coverage by increasing λ. In fact, there is statistical significance (p-value below 0.05) only in the case of `calday` and `sa`. This small influence can be due to the intrinsic high coverage obtained by the ES in all the test programs. The number of evaluations (and the time) is, in general, increased with the number of offsprings, because in each step of the ES there are more individuals to evaluate. The number of evaluations and the time are statistically significant for all the test programs except for `gcd`, `insertion`, and `shell`. From the results we conclude that $\lambda = 1$ seems to be the best number of offsprings.

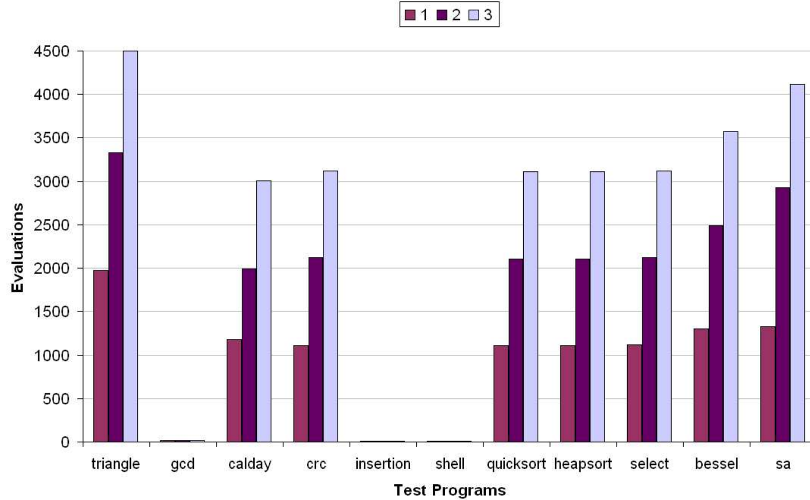

Fig. 5. Number of evaluations for λ=1,2,3. The value of μ is set to 10.

Let us now turn to study the influence of the population size μ. We fix the number of offsprings to the best previous value $\lambda = 1$. In Table 5 and Fig. 6 we present the results.

Table 5. Results after changing the size of the population μ.

Test	(1+1)-ES		(5+1)-ES		(10+1)-ES		(20+1)-ES		(30+1)-ES	
Programs	Cor.(%)	Evals.	Cor.(%)	Evals.	Cor.(%)	Evals.	Cor.(%)	Evals.	Cor.(%)	Evals.
triangle	35.77	1020	100.00	2010	99.92	1975	99.76	2178	99.84	2468
gcd	81.33	1020	100.00	21	100.00	21	100.00	30	100.00	39
calday	78.48	1020	93.03	1070	91.82	1182	91.52	1268	91.36	1386
crc	99.61	1020	100.00	1064	100.00	1114	100.00	1243	100.00	1330
insertion	100.00	10	100.00	10	100.00	10	100.00	10	100.00	10
shell	100.00	10	100.00	10	100.00	10	100.00	10	100.00	10
quicksort	94.12	1020	94.12	1060	94.12	1110	94.12	1210	94.12	1310
heapsort	100.00	1020	100.00	1060	100.00	1110	100.00	1210	100.00	1310
select	76.21	1020	83.33	1065	83.33	1120	83.33	1226	83.33	1338
bessel	80.16	1020	97.56	1233	97.40	1306	97.48	1446	97.56	1535
sa	96.61	1020	98.81	1309	98.70	1329	98.98	1513	99.04	1683

In this case the coverage is approximately the same for a population size larger than one. In all the test programs the corrected coverage of the (1+1)-ES is statistically significant with the obtained by the other ESs. However, there is no statistical significance in the corrected coverage among the rest of the ESs except for `calday` in (5+1)-ES versus (30+1)-ES. With respect to the number of evaluations, it increases with the population size because there are more individuals in the initial population to evaluate. However, we observe an exception in (1+1)-ES for `gcd`: with a population of 1 individual the ES can not get 100% coverage and this explains the high number of evaluations needed.

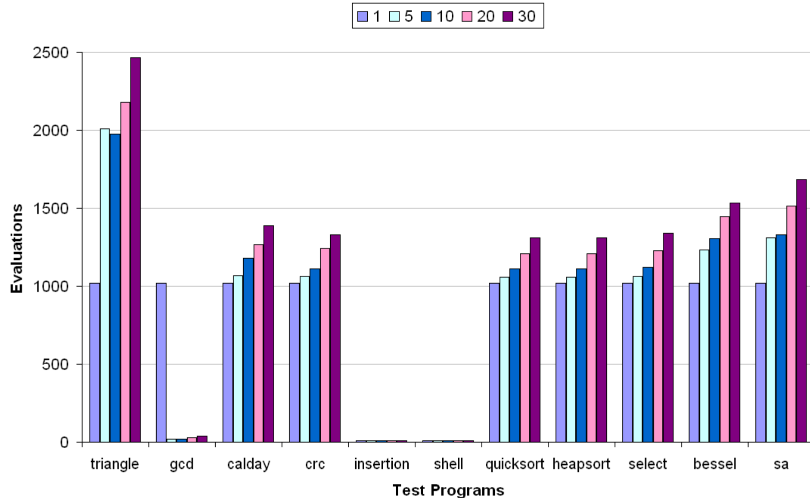

Fig. 6. Number of evaluations for μ=1,5,10,20,30. The value of λ is set to 1.

As a global conclusion, we can state that the number of offsprings (λ) has a small influence in the coverage, but an important influence in the number of evaluations and, as a consequence, in the time. With respect to the population size (μ), we observe that the coverage is approximately the same when the population size is above a few individuals. The use of a single individual in the population must be avoided if the coverage is an important factor in the optimization. In general, the number of evaluations is increased with the size of the population. However, the growth is moderate in comparison with that provoked by changing λ. In order to end this discussion, we conclude that the best parameterization of ES for this benchmark is $\mu = 5$ and $\lambda = 1$.

4.3 A Comparison Between ES and GA

In this section we compare our (10+1)-ES with a Genetic Algorithm. A GA is another kind of evolutionary algorithm that usually represents the solutions by means of a binary string (chromosome). However, other representations have been employed with GAs, so we can not use this feature as a distinguishing difference. In fact, the individuals we use in our GA are not binary strings, they are strings of numeric values and they can be integer or real numbers. These numbers are the test program inputs. This algorithm does not use any self-adaptive parameter in the individual as the ES does, the individual contains only the representation of a solution. In opposite to the ES, the recombination operator has a great importance in GAs. We use here the double point crossover (DPX) that works as follows: it chooses two points in the parent strings and swaps all the numeric values between them. The mutation operator perturbs all numbers by adding a random value. The probability distribution of these random values

is a normal distribution with mean zero and standard deviation one. The GA has ten individuals in the population and generates one only offspring by recombination and mutation (steady state GA). The maximum number of iterations is 100 (110 evaluations), like in the ES. The 10% of the initial population (one individual) is seeded by using the mechanism explained in Sect. 2. Results are summarized in Table 6. We highlight in boldface the best results.

Table 6. Comparison between ES and GA.

Test	**(10+1)-ES**		**GA**	
Programs	Cor. cov. (%)	Evals.	Cor. cov. (%)	Evals.
triangle	**99.92**	**1975**	95.20	2009
gcd	**100.00**	**21**	**100.00**	252
calday	**91.82**	**1182**	90.91	1217
crc	**100.00**	**1114**	**100.00**	1121
insertion	**100.00**	**10**	**100.00**	**10**
shell	**100.00**	**10**	**100.00**	**10**
quicksort	**94.12**	**1110**	**94.12**	**1110**
heapsort	**100.00**	**1110**	**100.00**	**1110**
select	**83.33**	**1120**	**83.33**	1339
bessel	**97.40**	**1306**	96.91	1557
sa	**98.70**	1329	96.61	**1110**
Average	96.84	935	96.10	986

We observe in Table 6 that the coverage is higher for the ES than the GA in the `triangle` program. This difference is statistically significant (p-value $\approx 10^{-9}$). For the `gcd` program the two algorithms get the maximum coverage. However, the GA needs 252 evaluations (that is, several iterations of the main loop) to get this coverage while the ES only needs 21 evaluations (that is, one iteration). In this case the ES has reached the objective very fast while the GA needs a larger effort. The observed advantage of the ES for the `calday` program with respect to the coverage is almost statistically significant (p-value 0.051, what means a 94.9% of confidence).

For the next six test programs (from `crc` to `select`) the two algorithms get the same coverage and similar number of evaluations with a slight advantage of the ES in two of them. At this point we can also conclude that `insertion` and `shell` are the easiest programs for the two test generators (with ES and GA). They only need 10 program executions to reach the objective.

The ES gets higher coverage than the GA for the two last programs (`bessel` and `sa`). However, the difference is statistically significant only in the case of `sa`. With respect to the number of evaluations, the `sa` program is the only test program in the table requiring a higher number of evaluations from the ES with respect to the GA.

As a global consideration, we observe from Table 6 that there is only a single value not highlighted in the ES columns, that is, all the results obtained with

the ES are equal or better than those obtained by the GA. In addition, the ES has less parameters to fix than the GA (only population size and number of offsprings) and, for this reason, ES is easier to use than GA.

4.4 Previous Results

The task of comparing our results against previous works is very hard. First, we need to find works tackling the same test programs. There is a test program very popular in the domain of software testing: the triangle classifier. However, there are several different implementations in the literature and, usually, the source code is not provided. In [2] the source code of the triangle classifier is published in the paper. In this work, we use that implementation for making performance comparisons. We have two other test programs in common with [2]: the computation of the greatest common denominator and the insertion sort. However, we use different implementations for these algorithms. In order to ease future comparisons we have indicated in Table 2 how to get the source code of the test programs (available at URL `http://tracer.lcc.uma.es`).

A second obstacle to compare different techniques is that of the measurements. We use the average condition-decision coverage and a corrected coverage to measure the quality of the solutions. In [7, 14, 15] the authors report only on the branch coverage. On the other hand, the coverage measurement used in [2] is obtained by using a proprietary software: DeepCover. We can not directly compare all these results in a quantitative manner because all the works use different measurements. Another obstacle is the number of independent runs, that may affect the results. A low number of independent runs is not enough to obtain a clear idea about the behavior of the technique employed. It is not our case, but some papers perform a low number of independent runs.

In spite of all the previous considerations we include in Table 7 the best average coverage results reported for the triangle classifier algorithm in [2, 7, 14, 15] and their needed number of evaluations (program tests). We show in the same table the results of our (5+1)-ES, the best in coverage percentage.

Table 7. Previous results of coverage and number of evaluations for `triangle`.

`triangle`	Ref. [7]	Ref. [2]	Ref. [14]	Ref. [15]	(5+1)-ES (here)
Coverage (%)	100.00[a]	94.29[b]	100.00[a]	100.00[a]	100.00[c]
Evaluations	18000	≈ 8000	608	3439	2010

[a] Branch coverage.
[b] DeepCover coverage.
[c] Corrected condition-decision coverage.

If we focus on the coverage of the Table 7 our (5+1)-ES has the best behavior at the same time as the algorithms of [7, 14, 15]. Comparing the number of evaluations we find the best (lowest) value in the work of Sagarna and Lozano [14]

followed by our (5+1)-ES. However, we must recall that the condition-decision coverage (used in this work) is a test adequacy criterion harder than the branch coverage used in [14] (see Sect. 2).

5 Conclusions

In this article we have proposed the use of an Evolutionary Strategy to solve the test data generation problem in computer software. We employed a benchmark with eleven test programs implementing some fundamental algorithms in computer science.

We have studied different parameterizations for the ES, in particular we have analyzed the influence of the number of offsprings and the population size in the canonical algorithm. The former does not affect significatively the coverage, but the number of evaluations. The population size has a negligible influence in the coverage when it is larger than one individual, but the number of evaluations increases with it. We therefore propose the (5+1)-ES as the clear best algorithm in this study from the points of view of accuracy (coverage) and efficiency (evaluations). In fact, our (5+1)-ES outperforms the results of other works in coverage percentage and number of evaluations.

Comparing the (10+1)-ES with the GA we observe a clear advantage of the ES in accuracy and efficiency. Furthermore, ES has less algorithm parameters to fix than GA, what means that the user can tune the system faster and does not require too much knowledge about it. This is a very good feature for those people interested in automatic software testing but with a little knowledge about Evolutionary Computation.

As future work, we plan to apply the ES-based test generator to other programs. We are specially interested in telecommunication software such as protocols, routing algorithms, and so on. In addition, we want to perform deeper improvements and use other metaheuristic techniques.

Acknowledgements

This work has been partially funded by the Ministry of Science and Technology (MCYT) and Regional Development European Found (FEDER) under contract TIC2002-04498-C05-02 (the TRACER project, `http://tracer.lcc.uma.es`). F. Chicano is supported by a grant from the Junta de Andalucía.

References

1. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering **16** (1990) 870–879
2. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Transactions on Software Engineering **27** (2001) 1085–1110
3. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering **2** (1976) 215–222

4. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Trans. Software Eng. **2** (1976) 223–226
5. Bird, D., Munoz, C.: Automatic generation of random self-checking test cases. IBM Systems Journal **22** (1983) 229–245
6. Offutt, J.: An integrated automatic test data generation system. Journal of Systems Integration **1** (1991) 391–409
7. Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Software Engineering Journal **11** (1996) 299–306
8. Wegener, J., Sthamer, H., Jones, B.F., Eyres, D.E.: Testing real-time systems using genetic algorithms. Software Quality Journal **6** (1997) 127–135
9. Mantere, T., Alander, J.T.: Evolutionary software engineering, a review. Applied Soft Computing **5** (2005) 315–331
10. Ostrowski, D.A., Reynolds, R.G.: Knowledge-based software testing agent using evolutionary learning with cultural algorithms. In: Proceedings of the Congress on Evolutionary Computation. Volume 3. (1999) 1657–1663
11. Tracey, N.: A search-based automated test-data generation framework for safety-critical software. PhD thesis, University of York (2000)
12. Tracey, N., Clark, J., Mander, K., McDermid, J.: An automated framework for structural test-data generation. In: Proceedings of the 13th IEEE Conference on Automated Software Engineering. (1998) 285–288
13. Díaz, E., Tuya, J., Blanco, R.: Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03), Montreal, Quebec, Canada (2003) 310–313
14. Sagarna, R., Lozano, J.A.: Variable search space for software testing. In: Proceedings of the International Conference on Neural Networks and Signal Processing. Volume 1., IEEE Press (2003) 575–578
15. Sagarna, R., Lozano, J.A.: Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. European Journal of Operational Research (2005) (in press).
16. Sthamer, H., Wegener, J., Baresel, A.: Using evolutionary testing to improve efficiency and quality in software testing. In: Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis & Review, Melbourne, Australia (2002)
17. Bäck, T.: Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Oxford University Press, New York (1996)
18. Rechenberg, I.: Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Fromman-Holzboog Verlag, Stuttgart (1973)
19. Rudolph, G.: 9. In: Evolutionary Computation 1. Basic Algorithms and Operators. Volume 1. IOP Publishing Lt (2000) 81–88