On the performance of Estimation of Distribution Algorithms applied to Software Testing

Ramón Sagarna, Jose A. Lozano*

Department of Computer Science and Artificial Intelligence

University of the Basque Country

Paseo Manuel Lardiazabal 1

20018 San Sebastián, Guipuzcoa, Spain

E-mail: {ccbsaalr,ccploalj}@si.ehu.es

26th March 2003

*This work was supported by the Basque Government and by the Ministry of Science and Technology under grants UE2000-16 and TIC2001-2973-C05-03 respectively. R. Sagarna was also supported by the Department of Education, Universities and Research of the Basque Government under the programme of Formation of Researchers. The authors also wish to thank Joachim Wegener and Pedro Larrañaga for the help provided.

Abstract

One of the most important issues in software testing is the generation of the input cases used during the test. Due to the expensive cost of this task its automatization has become a key aspect. An alternative to obtain this is by evolutionary testing. The aim of evolutionary testing is the creation of test data by means of combinatorial optimization search methods.

A heuristic approach to the automatic generation of test cases is presented. The developed approach makes use of an emerging set of evolutionary algorithms called Estimation of Distribution Algorithms. The analysis of the experimental results obtained shows this set of optimization techniques as a promising option for tackling this problem. More precisely, the performance of different Estimation of Distribution Algorithms is evaluated and a comparison with regard to the results of other works is carried out.

Keywords: evolutionary testing, test case generation, branch coverage, evolutionary algorithms, estimation of distribution algorithms

1 Introduction

Testing is a crucial part of the software development process (Beizer, 1990). It plays a main role in the search for the required quality as it constitutes the primary way used in practice to verify the correct behaviour of the produced software. In fact, it is not unusual to dedicate at least 50% of the project resources to this phase.

A major activity in testing is the test design. From this activity, a key task is the generation of the input cases to be applied to the programme under test. This is a critical task, as the produced input must be adequate to the test type and its requirements. This, together with the fact that in most of the organizations this generation is performed manually, results in a high amount of resources dedicated to the test data generation step. Thus, the automatic generation of test cases is desirable and a number of authors (Ould, 1991) suggest it is vital for the software testing area.

Existing methods for test data generation mainly follow three strategies: random, functional and structural. Random testing samples input domain according to a probability distribution. Due to its simplicity, a common approach is automatically generating test cases simulating an uniform distribution (Duran and Ntafos, 1984). Alternatively, in functional testing, the inputs are obtained from the programme specification. That is, they are generated taking the functional properties of the programme into account. Its automatization demands a formal specification, e.g. in (Burton, 2000) a framework orientated to a popular specification language named Z was described.

The most used test data generation method is the structural testing strategy, which is based on the internal structure of the programme. The source code reveals control or data flow entities such as the branches the flow of control can take from a conditional statement or the different possible usages of a variable. A structural generator aims at creating test inputs that exercise these entities. More precisely, entities define several adequacy criteria which generators search to fulfil. For instance, branch coverage is a classical criterion stating that every programme branch must be exercised. Other typical criteria are passing through every code statement (statement coverage), exercising every sequence of branches from the programme input to its output (path coverage), or covering all the pairs definition-usage for each variable (all-defs coverage). Regardless of the adequacy criterion, there is a need to know the level of completion attained by the test case generator. This is what the coverage measurement indicates, i.e. the percentage of structural entities exercised for a given criterion. In practice, coverage measurements and structural criteria are also used by many methods from other testing strategies and vice versa.

Generally, automated structural testing is reached by means of static or dynamic test data generation. Static methods are commonly based on symbolic execution. This technique consists of choosing an entity from a graph representing the programme structure, and building a system of inequalities by

assigning symbolic values to variables affecting the entity and respecting the constraints associated to the conditions in the code. A solution to the system is an input exercising the selected entity. In (Demillo and Offut, 1993), a work using this method can be consulted. On the other hand, the underlying idea in dynamic methods is addressing automatic generation of test data as an optimization problem. An instrumented version of the programme is constructed, that is, the programme is expanded with instructions that will extract information concerning the execution of an input. The collected information is used to assess the closeness of the executed inputs to cover the desired structural entities and guide the search towards new inputs to be executed. In (Korel, 1990), the obtained information determined a function value assigned to each input after execution. The objective was to find an input minimizing its function value, which only occured when reaching the target entity.

A field which is arousing the interest of researchers is evolutionary testing. The aim of evolutionary testing is to seek for test cases employing modern heuristic combinatorial optimization techniques during the process, therefore, generalizing the optimization point of view in dynamic test data generation to any testing strategy. Promising results have already been attained in the field for structural adequacy criteria (Jones et al., 1996) (McGraw et al., 1998) (Wegener et al., 2001).

The present work describes an evolutionary testing approach for the structural automatic generation of test cases. The proposed method deals with the branch coverage adequacy criterion through a two step process, selection of the next branch to exercise and search for an input reaching that branch. This search step constitutes the major difference with respect to other works from the field as, for the first time, it is carried out through Estimation of Distribution Algorithms (EDAs) (Larrañaga and Lozano, 2002) (Mühlenbein and Paaß, 1996).

EDAs are a new set of evolutionary algorithms which actually deserves the attention of the evolutionary computation community. These algorithms replace the classical crossover and mutation operators by an estimation of the probability distribution associated to the selected individuals and the sampling of this distribution to create the next population. The use of this optimization technique for test case generation comes motivated by the fact that it has already been applied to other problems with excellent results (Inza et al., 2000). More precisely, EDAs required less generations to obtain the same quality results as other evolutionary algorithms.

In order to prove the alternative here exposed, several experiments were conducted by considering different EDA instances. The results are analysed, obtaining encouraging conclusions about the choice of EDAs to solve the software testing problem. The analysis involves two aspects, a comparison of the obtained results with those of other works and an evaluation of the behaviour of the different EDA instances. With regard to the former, in most of cases the same coverage is reached generating less inputs during the process. Related to the second aspect, EDAs are compared with each other and the most adequate is then identified.

```
Repeat until stopping criterion met
    E ← Select entity to exercise
    Obtain input minimizing function for E
```

Figure 1: Test case generation scheme.

The rest of the paper is organized as follows. The next section reviews evolutionary testing and a few salient references from this field. Section 3 makes a general introduction to EDAs and briefly presents a number of them. The developed approach in this work is described in Section 4. Next, the experiments carried out and their results analysis are shown. Finally, obtained conclusions and future work are included.

2 Evolutionary Testing

Evolutionary testing alludes to software test data generation making use of modern heuristic combinatorial optimization techniques. Typical examples of such techniques are Simulated Annealing (Kirkpatrick et al., 1983), Tabu Search (Glover and Laguna, 1997), and evolutionary algorithms (Bäck, 1996). The idea of employing this type of optimization methods has also been applied in the testing of VLSI circuits (Corno et al., 1996).

The manner in which the heuristic technique takes part remains open, so evolutionary testing generalizes the optimization point of view in the dynamic test data generation alternative from structural testing to any other testing approach. However, most of the work on the field to date has concentrated on temporal behaviour testing and structural test data generation, whilst the usual heuristic optimization method has been the Genetic Algorithm (GA) (Goldberg, 1989).

Temporal behaviour testing tries to verify that execution timing constraints are satisfied. In (Pohlheim and Wegener, 1999), the presented automated approach relied entirely on a GA. Each individual represents an input given to the programme, and the fitness value is obtained by ranking the individuals execution times. The test runs until the GA reaches a maximum number of generations or until an optimum is found, i.e. an input violates the timing constraints.

Apropos of structural testing, many existing works are based on the system described in (Korel, 1990). This system conforms to a dynamic test data generation strategy where, at each step, an entity is selected as the actual goal and a function is associated. The function is formulated so that if an executed input exercises the goal, the value is minimum, otherwise the value is proportional to a closeness measurement obtained from the information given by the instrumented programme. Then, a simple function minimization technique is applied to find an input covering the goal entity.

Evolutionary testing approaches following the idea of the previous system come motivated by the fact that the space defined by inputs and goal is generally large and complex. Simple local search methods perform poorly in such spaces as they easily fall into local optima, therefore, more sophisticated search techniques become a suitable alternative.

Thus, developed works consist basically on achieving an adequacy criterion through the iterative process shown in Figure 1. At each round, the coverage of an entity is tried by means of two consecutive phases, selection and optimization.

Although different strategies exist for selection, the common trend is to determine the objective entity with the help of a graph that reflects the structural characteristics of the programme. For example, in the case of branch coverage, a control flow graph (Fenton, 1985) is usually employed. To obtain a control flow graph from the programme, every basic block in the code must be identified, i.e. every maximal sequence of code statements such that if one is executed then all do. In a control flow graph (G, s, e), where $G = (X, U)$, the two unique vertices s and e represent the programme entry and exit, while every other vertex represents a code basic block. An arc $(x, y) \in U$, with x and y distinct from s and e, is such that the control of the programme can be transfered from the block x to the y without crossing any other block. Analogously, for every arc $(s, x) \in U$ or $(y, e) \in U$, it will be possible to transfer the flow of control from the entry to block x and from block y to the exit respectively. In this kind of graph, programme branches will come defined by every vertex x with $outdegree(x) > 1$.

The next phase of the scheme in Figure 1 tackles an optimization problem, to be precise, given the search space Ω formed by the programme inputs and a function $h : \Omega \longrightarrow \mathbb{R}$, find $\boldsymbol{x}^* \in \Omega$ such that $h(\boldsymbol{x}^*) \leq h(\boldsymbol{x})\ \forall \boldsymbol{x} \in \Omega$. Obviously, the function h must take its minimum value when an input reaches the objective entity. This optimization problem is addressed with an evolutionary algorithm.

Typically, the objective function is created according to the following strategy. Given an expression $\mathcal{A}\,\mathbf{OP}\,\mathcal{B}$ of a conditional statement **COND** in the code, with **OP** denoting a comparison operator, the value for an executed input $\boldsymbol{x}$ is determined by the following function:

$$h(\boldsymbol{x}) = \begin{cases} l & \text{if } \mathbf{COND} \text{ not reached} \\ d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}}) + K & \text{if } \mathbf{COND} \text{ reached and objective not attained} \\ 0 & \text{otherwise} \end{cases}$$

where l is the highest computable value, $\mathcal{A}_{\boldsymbol{x}}$ and $\mathcal{B}_{\boldsymbol{x}}$ are an appropriate representation of the values taken by $\mathcal{A}$ and $\mathcal{B}$ in the execution, $d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}})$ is a distance measurement between $\mathcal{A}_{\boldsymbol{x}}$ and $\mathcal{B}_{\boldsymbol{x}}$, and K is a previously defined constant. Commonly, if $\mathcal{A}$ and $\mathcal{B}$ are numerical, then $\mathcal{A}_{\boldsymbol{x}}$ and $\mathcal{B}_{\boldsymbol{x}}$ are their values and $d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}}) = |\mathcal{A}_{\boldsymbol{x}} - \mathcal{B}_{\boldsymbol{x}}|$. In the case of more complex data types, a binary representation of the values for $\mathcal{A}$ and $\mathcal{B}$ can be obtained and, for instance, let $d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}})$ be the Hamming distance (Sthamer, 1996).

In the case of a compound expression, the overall objective function is constructed from the partial functions for each subexpression. Given two subexpressions C_1 and C_2 with their respective functions h_1 and h_2, and an input $\boldsymbol{x}$, the value for the logical expression $C_1 \vee C_2$ is $min\{h_1(\boldsymbol{x}), h_2(\boldsymbol{x})\}$, the logical expression $C_1 \wedge C_2$ is calculated as $h_1(\boldsymbol{x}) + h_2(\boldsymbol{x})$, and for $\neg C_1$ the value is known by propagating the negation inside C_1. By associating different logical expressions, the overall value for h is obtained.

In (McGraw et al., 1998) condition/decision coverage for C/C++ programmes is sought, that is, every branch needs to be taken and each expression in each conditional statement must be both true and false. Executed inputs are stored and the selection phase consists of choosing an entity for which there are inputs reaching its associated conditional statement. Notice that, unless the criterion is fulfilled, such an entity always exists as after every execution, there will be a branch exercised that its sibling was not. Once the objective is fixed, the minimum of a function of the form previously described is pursued using a GA where an individual represents an input. In (Jones et al., 1996) and (Sthamer, 1996), the same type of objective function is applied with a GA for branch coverage. Simulated Annealing is proposed in (Tracey et al., 1998). The function is smoothed in (Wegener et al., 2001) taking a level of proximity into account for the inputs not reaching the conditional statement of the branch objective, again GAs are used as the search method. In contrast, in the approach presented in (Pargas et al., 1999) the function is just the proximity measurement.

Other works do not follow the scheme in Figure 1. The developed system in (Roper et al., 1995) consisted of a GA where each individual represented an input. The fitness function was the coverage gained by the individual, and the algorithm evolved for a maximum number of generations or until a set of individuals from the population obtained the desired coverage. An alternative to all the previous approaches is explained in (Smith and Fogarty, 1996). Once again, a GA is used but, in this case, an individual corresponds to a set of test inputs, and the fitness is the coverage reached by the set after execution. This way, the problem of generating a set of test cases to fulfil an adequacy criterion is tackled from a pure evolutionary computation view, where an individual represents a solution to the whole problem.

3 Estimation of Distribution Algorithms

One of the most popular methods from evolutionary computation is the GA (Goldberg, 1989). GAs have been widely studied and applied to many problems with excellent results, however they suffer from some well-known weaknesses: (1) the determination of suitable values for the components and parameters needed by the algorithm (crossover and mutation operators, probabilities of crossover and mutation, population size, rate of generational reproduction, number of generations, etc.) can lead itself to an optimization problem (Grefenstette, 1986), (2) there are problems - deceptive problems -

where GAs fail and offer a poor behaviour, (3) relations between variables defining the individuals are kept implicitly in the building blocks.

All these obstacles motivated the birth of a new type of population based evolutionary algorithm: Estimation of Distribution Algorithms (EDAs) (Larrañaga and Lozano, 2002) (Mühlenbein and Paaß, 1996). In the EDA approach, there are neither crossover nor mutation operators. The individuals forming the population of the next generation are obtained sampling a probability distribution previously estimated from the selected individuals. In contrast to other evolutionary computation heuristics such as GAs, in EDAs relations between the variables of individuals are explicitly expressed by the joint probability distribution estimated from the set of selected individuals.

3.1 The abstract EDA

Following concepts refer to discrete domains, that is, variables of individuals are discrete. For a more extended description refer to (Larrañaga and Lozano, 2002) and (Pelikan et al., 1999), in (Larrañaga and Lozano, 2002) continuous domains are also included.

First, some notation which will be used in the rest of the section is required. Given an n-dimensional random variable $\boldsymbol{X} = (X_1, X_2, ..., X_n)$ and a possible instantiation $\boldsymbol{x} = (x_1, x_2, ..., x_n)$, the joint probability distribution of $\boldsymbol{X}$ will come denoted by $p(\boldsymbol{x}) = p(\boldsymbol{X} = \boldsymbol{x})$. In the case of two unidimensional random variables X_i, X_j and their respective possible values x_i, x_j, the conditional probability of X_i given X_j will be represented as $p(x_i|x_j) = p(X_i = x_i|X_j = x_j)$. In the context of evolutionary algorithms, an individual of length n is an instantiation $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ of $\boldsymbol{X} = (X_1, X_2, ..., X_n)$. The population of the l-th generation will be written D_l and the selected individuals D_l^{Se} constitute a data set of N cases of $\boldsymbol{X} = (X_1, X_2, ..., X_n)$. EDAs estimate $p(\boldsymbol{x})$ from D_l^{Se}, i.e. it is the probability of an individual being among the selected individuals. This estimation must be carried out at each generation, therefore, the joint probability distribution of the l-th generation will be represented by $p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{Se})$.

A pseudocode for the abstract EDA is presented in Figure 2. The selection step can be done using any of the strategies in evolutionary computation, e.g. rank-based selection. Once the individuals are selected they are treated as a data base from which their probability distribution is estimated. Then this distribution is simulated to create the individuals of a new population. These steps are repeated until a previously defined stopping criterion is met.

The key point of EDAs is how the probability distribution is estimated at each generation. The computation of all the parameters for $p(\boldsymbol{x})$ is unviable, therefore, it is factorized according to a probability model that in some cases, limits the possible dependencies among the variables $X_1, X_2, ..., X_n$. This leads to several approximations assuming different levels of complexity in their models. The alter-

$Do \leftarrow$ Generate M individuals (the initial population) randomly

Repeat for $l = 1, 2, ...,$ until stopping criterion met

$D_{l-1}^{Se} \leftarrow$ Select $N \leq M$ individuals from D_{l-1}

$p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{Se}) \leftarrow$ Estimate the probability distribution of an individual being among the selected individuals

$D_l \leftarrow$ Sample M individuals (the new population) from $p_l(\boldsymbol{x})$

Figure 2: Pseudocode for abstract EDA.

natives range from those where the variables are mutually independent, to those with no restrictions to the variables interdependencies. The most restrictive models prevent from the induction of probability distributions having dependencies. However, they allow for a fast and easy estimation that may convert them in a suitable possibility for solving a problem. On the other hand, the least restrictive models are able to show all the dependencies between the variables in a problem although their computational cost is expensive and for some cases can result in an inpractical choice.

Next, some EDA approaches are briefly presented. Taking into account the probabilistic model complexity, they are classified as univariate, bivariate and multivariate EDAs. For a more detailed description, including other EDAs, refer to (Larrañaga and Lozano, 2002) and (Pelikan et al., 1999).

3.2 Univariate EDAs

These type of EDAs assume that the n-dimensional joint probability distribution is decomposed as a product of n univariate independent probability distributions, i.e.

$$p_l(\boldsymbol{x}) = \prod_{i=1}^{n} p_l(x_i).$$

3.2.1 UMDA

Univariate Marginal Distribution Algorithm (UMDA) (Mühlenbein, 1998) is one of the most simple EDAs. It estimates $p_l(x_i)$ as the relative frequencies of x_i in the data set D_{l-1}^{Se}. That is:

$$p_l(x_i) = \frac{\sum_{j=1}^{N} \delta_j(X_i = x_i|D_{l-1}^{Se})}{N}$$

where $\delta_j(X_i = x_i|D_{l-1}^{Se}) = \begin{cases} 1 & \text{if } X_i = x_i \text{ in the } j\text{-th case of } D_{l-1}^{Se} \\ 0 & \text{otherwise.} \end{cases}$

3.2.2 PBIL

Population Based Incremental Learning (PBIL) (Baluja, 1994) is a more elaborated algorithm. It was developed upon the idea of dealing with binary search spaces, therefore, the variables $X_1, X_2, ..., X_n$

are dichotomic. Thus, in this special case $p_l(x_i)$ will represent the probability of X_i taking a value of 1 in the l-th generation. After selection of the N individuals, $p_l(x_i)$ is obtained updating the stored $p_{l-1}(x_i)$ by means of a Hebbian inspired rule:

$$p_l(x_i) = (1-\alpha)p_{l-1}(x_i) + \alpha\frac{1}{N}\sum_{j=1}^{N} x_i^j$$

where x_i^j is the value of X_i for the j-th case in D_{l-1}^{Se}, and $\alpha \in (0,1]$ is a learning rate previously specified.

3.3 Bivariate EDAs

These approximations make use of second order statistics in order to estimate the joint probability distribution. Apart from the probability values, a structure that reflects the dependencies must be learnt. The factorization carried out by the models from this category can be expressed as follows:

$$p_l(\boldsymbol{x}) = \prod_{i=1}^{n} p_l(x_i|x_{j(i)})$$

where $X_{j(i)}$ is the variable, if any, on which X_i depends.

3.3.1 MIMIC

Mutual Information Maximization for Input Clustering (MIMIC) (De Bonet et al., 1997) is based on a model that factorizes the probability distribution in the following way:

$$p_l(\boldsymbol{x}) = p_l(x_{i_1}|x_{i_2}) \cdot p_l(x_{i_2}|x_{i_3}) \cdot ... \cdot p_l(x_{i_{n-1}}|x_{i_n})$$

where $i_1, ..., i_n$ represents a permutation of the numbers between 1 and n.

The algorithm searches for the permutation that minimizes the Kullback-Leibler divergence of $p_l(\boldsymbol{x})$ with respect to the empirical probability distribution associated to D_{l-1}^{Se}. To find this permutation, a greedy search is carried out.

3.3.2 TREE

TREE (Larrañaga and Lozano, 2002) refers to an adaptation of the Combining Optimizers with Mutual Information Trees (COMIT) algorithm (Baluja and Davies, 1997). The COMIT algorithm hybridizes the EDA approach with local search methods, while TREE only involves the EDA part. The estimation of the probability distribution is carried out using the Maximum Weight Spanning Tree (MWST) algorithm (Chow and Liu, 1968), which is based on the construction of a tree structure. Initially, the weight of each (X_i, X_j) branch is calculated as the mutual information measurement between X_i and X_j. Then, candidate branches are increasingly added in the tree. Once the structure is terminated, $p_l(\boldsymbol{x})$ is obtained through the conditional relative frequencies in D_{l-1}^{Se}.

Notice that in this model, two distinct variables may depend on the same, while in MIMIC this cannot happen because of the permutation.

3.4 Multivariate EDAs

With the models from this category, the expression of all the existing variable interdependencies in a problem is possible. The probability distribution is estimated by means of probabilistic graphical models (Castillo et al., 1997), which use a graph to represent the detected interdependencies between the variables. Thus, the factorization associated to this type of EDAs is as follows:

$$p_l(\boldsymbol{x}) = \prod_{i=1}^{n} p_l(x_i|\boldsymbol{pa_i})$$

where $\boldsymbol{pa_i}$ are the instantiations of $\boldsymbol{Pa_i}$, the set of variables on which X_i depends.

3.4.1 EBNA

In the Estimation of Bayesian Network Algorithm (EBNA) (Larrañaga et al., 2000), the factorization of the joint probability distribution is given by a Bayesian network (BN) learnt from D_{l-1}^{Se}. A BN is a pair $(S, \boldsymbol{\theta})$ where S is a directed acyclic graph representing the (in)dependencies between the variables and $\boldsymbol{\theta}$ is the set of conditional probability values needed to define the joint probability distribution.

The method used to learn the structure S leads to different EBNA instantiations, for example, EBNA_{BIC} and $\text{EBNA}_{\text{K2+pen}}$ seek the structure with the best BIC (Schwarz, 1978) and penalized K2 (Cooper and Herskovits, 1992) score metric respectively.

Regardless of how the BN is learnt, EBNA simulates $p_l(\boldsymbol{x})$ by means of Probabilistic Logic Sampling (Henrion, 1988).

4 Estimation of Distribution Algorithms in Evolutionary Testing

Until now, approaches in the evolutionary testing literature have mainly concentrated on the use of GAs and slightly on Simulated Annealing. However, many other heuristic optimization methods can be exploited. The alternative here described makes use of EDAs to deal with branch coverage for C/C++ programmes.

The system conforms to the general scheme in Figure 1. Selection phase follows the proposed option in (Wegener et al., 2001), while optimization step allows the application of several EDAs. Each code branch is associated with three possible states, covered, treated but uncovered or untreated.

The stopping criterion of the scheme is full coverage achievement or unsuccesful treatment of every unexercised objective branch.

4.1 Optimization phase

This phase tries to solve the optimization problem raised in Section 2 by means of an EDA. Each individual is a 0-1 string representing an input, and its fitness is the value taken by function h. Implementation of h is based on the function type also described in Section 2. Information needed to calculate the fitness value is obtained from the performed instrumentation on the programme. This instrumentation returns the values of expressions for conditional statements during the execution of the input represented by the individual.

Each branch is bound with a set of individuals which is used as the seed population for the EDA when the branch is the objective. Thus, although the objective branch is fixed, each individual is evaluated according to every other branch. If the branch is exercised, its state is marked as covered and the input is stored, otherwise if the fitness of the individual is better than the worst individual in the set associated to the branch, then this last is replaced in the set by the new better individual and, if the branch had previously been treated, its state is marked as untreated. This notion of seeding has also been exploited in other works (McGraw et al., 1998) (Wegener et al., 2001). Figure 3 shows the evaluation algorithm for an individual $\boldsymbol{i}$, the value of function h associated to branch b for a given input $\boldsymbol{x} \in \Omega$ is represented by $h_b(\boldsymbol{x})$.

The EDA finishes when either the minimum is found, i.e. an input covering the objective branch is found, or a maximum number of generations is reached. In the first case, the objective branch state is marked as covered and in the second as treated but uncovered.

Any EDA approach can be applied, however multivariate EDAs seem to be more adequate to this problem than simpler alternatives as the existence of variable interdependencies appears to be evident. Generally, input parameters act over programme variables which in turn interact affecting other variables and so on, leading to non-linear combinations that determine the branches followed by the control flow.

As can be noted, at each generation, a probability distribution is learnt and used to generate new individuals, that is, the input space Ω is sampled according to a probability distribution. Therefore, the EDA follows a random testing strategy inside each generation. On the other hand, the probability distribution is obtained from the selected individuals, and these are chosen with respect to their fitness, i.e. a dynamic test data generation tactic is adopted. Consequently, this approach can be described as a hybrid between random and dynamic test data generation. This allows for the outlining of the behaviour of the refered method, from the testing perspective, as a random generation of inputs which,

```
x ← Translate individual i to input
Execute instrumented programme with x
Repeat for each uncovered branch b
    f_b^i ← h_b(x)
    f_b^w ← Find the fitness of the worst individual
    w in the set associated to b
    If f_b^i < f_b^x
        Substitute w by i in the set associated to b
        If f_b^i = 0
            Mark b as covered
        else
            Mark b as untreated
    If b is the objective
        fitness ← f_b^i
Return fitness
```

Figure 3: Pseudocode for the evaluation algorithm.

at each generation, updates its distribution on the basis of the already generated ones.

4.2 Selection phase

A control flow graph is used to identify the branches at the initialization stage and help to decide which branch to select next during the process. In a control flow graph, branches come defined by every vertex x with $outdegree(x) > 1$, that is, every arc (x, y) highlights a new branch.

Candidate objective branches are those in the untreated state. The branch objective will be the one for which the mean fitness of its associated individuals is best. In case a tie is given, then a breadth first search is carried out, i.e. from the tied branches, selection of the one with lowest level in the control flow graph. A pseudocode of the selection algorithm can be observed in Figure 4.

The idea is facing the optimization problem with the most promising population seed available at that moment. As one can see, it is possible for a branch already treated, to be a candidate objective once again if, during optimization, a new individual is introduced in its set. The reason for this is that mean fitness in the set is better than before optimization and could result in a promising population seed.

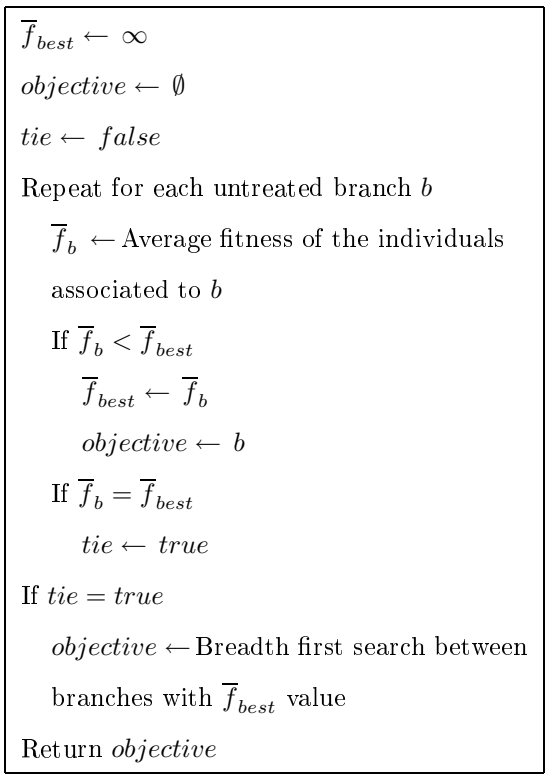

Figure 4: Pseudocode for the selection algorithm.

4.3 An execution example

As an illustration of the developed approach, some of the steps of a hypothetical system execution will be explained. Figure 5 presents a function, written in the C programming language, which returns the solution, if it exists, to a quadratic equation. The input consists of three integer parameters representing the equation coefficients.

First of all, the control flow graph is obtained and an instrumented version of the programme constructed; both are shown in Figure 5.

After an execution of the instrumented code, an output file contains, at each line, the traversed basic block (numbered as in the control flow graph) and, if the previous block had a conditional statement, the values of the (sub)expression(s) in it calculated according to the strategy of Section 2. As seen in the figure, the instrumentation uses three artificially created probe functions. At the beginning, the call to *probe_init* initializes the output file and a required memory vector. The *probe_dist* function calculates an appropriate distance measurement between the second and third parameters and stores the result in the position of the memory vector given by the first parameter. In this example, the distance will always be the absolute value of the difference between the parameters as they are numerical. The *probe_out* function writes, in the output file, a new line containing the basic block number given as a parameter and, if any, every (sub)expression value in the memory vector. An example of an output

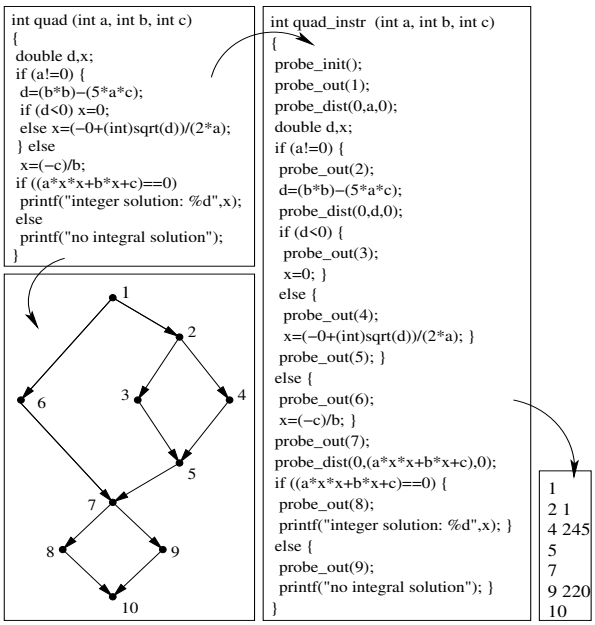

Figure 5: Example function, control flow graph, instrumented version and output file.

file is also presented in Figure 5.

In the system startup, programme branches are detected from the control flow graph and bound with a population of individuals created according to an uniform distribution.

Once this is done, the test case generation iterative process begins. Assuming the branch defined by the arc $(2, 3)$ is selected as the objective in a round, an EDA will pursue its coverage, i.e. the minimization of the function

$$h(\boldsymbol{x}) = \begin{cases} l & \text{if block 2 not reached} \\ |d - 0| + 1 & \text{if block 2 reached and block 3 not reached} \\ 0 & \text{otherwise} \end{cases}$$

where $\boldsymbol{x} = (a, b, c)$, l is the highest computable value and d is the value taken by the internal variable in the code during execution of $\boldsymbol{x}$.

Representing each integer parameter with 16 bits, an individual will be a 0-1 string of length 48. For the evaluation of the individual, this string is translated to a programme input which will be given to the instrumented programme for execution. Supposing the obtained input is $(1, 20, 31)$, then the instrumentation results are those of Figure 5. The fitness of the individual will be the value of h for the input. As the exercised branch was $(2, 4)$, in the line of block 4 the result of $|d - 0|$ is given, so the value of h is $245 + 1 = 246$. Although the individual is already evaluated, the performed execution is not discarded and the output file is used to calculate the function value for every branch distinct to the objective. If the value improves the fitness of the worst individual in the set associated to a branch,

then this worst individual is replaced by the actual one and the branch is marked as untreated.

The EDA will run in each objective branch until a maximum number of generations is reached or the minimum of h is found. In the first case, the branch objective is marked as treated and in the second as covered. Once the EDA finishes, another iteration of the overall process begins.

5 Experimental results

In order to observe how the presented approach performs in practice, several experiments were carried out. Overviewed EDAs in Section 3 are considered to generate test cases for a number of programmes extracted from the literature (McGraw et al., 1998) (Sthamer, 1996) (Wegener et al., 2001).

5.1 Experimental setting

The experiments involved seven classical programmes which are next explained.

Triangle

This is a popular programme in software testing experimentation. An input is composed of three numerical parameters, each representing the length of a segment. The aim is to detect the triangle type, if any, associated to the input. Four different versions were used. The Triangle1 programme (Wegener et al., 2001) owns three integer parameters which in the experiments took values in the interval $[-16384, 16383]$. Triangle2 (Wegener et al., 2001) is the same as Triangle1 with floating point parameters instead, the interval for each was $[-32768, 32768]$. On the other hand, Triangle3 (McGraw et al., 1998) is a new implementation with integer parameters for which the interval $[-512, 511]$ was chosen. Finally, Triangle4 (Sthamer, 1996) constitutes a distinct implementation once again, the selected interval for its integer parameters was $[-512, 511]$.

Atof

Given a string of characters as input, Atof (Wegener et al., 2001) transforms it into a floating point number if possible. For the experiments, the input string length was 10 characters codified with 7 bits each (the ASCII character set).

Remainder

This function (Sthamer, 1996) calculates the remainder of the division of two integer numbers, therefore an input is composed of two interger parameters for which the interval $[-32768, 32767]$ was chosen during experimentation.

Programme name	Branch number	Branch nesting depth	Myers interval
Triangle1	26	7	(14 : 21)
Triangle2	26	7	(14 : 21)
Triangle3	20	6	(11 : 18)
Triangle4	26	12	(14 : 14)
Atof	30	13	(16 : 41)
Remainder	18	5	(10 : 10)
Complexbranch	22	5	(11 : 23)

Table 1: Experimental programmes characteristics.

Complexbranch

In this case, there is no specific functionality as it is a function artificially created for testing purposes (Wegener et al., 2001). Its main characteristic is the existence of several hard to cover branches in the code. Six integer parameters form an input taking values in the interval $[-512, 511]$.

In order to have an idea about the programmes characteristics, Table 1 reflects the values for several structural complexity measurements (Shepperd, 1993). The branch number column shows the number of branches, i.e. number of objectives. Branch nesting depth points out the maximum nesting level for the branches in the code, that is, the maximum number of conditional statements must be fulfilled to cover a branch. In Myers interval, the lower bound is the number of conditional statements plus 1 and the upper bound the number of expressions of conditional statements plus 1, thus refering to the compound expressions.

However, care must be taken of misinterpreting these measurements as they do not represent the complexity of the programme with regard to branch coverage. According to the approach here exposed, the complexity of an objective branch attainment will come defined from two sources:

- the difficulty in reaching the associated conditional statement, if it is hard, then many individuals will take the value l as fitness and the landscape will be plateau like shaped,
- the function determined by the distance when the conditional statement is achieved.

The experiments involved all the EDAs introduced in Section 3; for the PBIL algorithm, the value of the α parameter was set to 0.5. For each EDA and each programme, four different population sizes as well as four values for the maximum number of generations were considered. Ten executions were run for each combination of these parameters.

pop.size	max. generations											
	50						100					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	3965	3205	3625	2945	3700	3960	5145	5480	4460	5035	7545	6665
	96.92	98.08	97.69	98.46	96.92	96.15*	97.69	97.31	98.08	97.31	96.54	96.92
100	6900	6530	5210	4360	6330	5540	6380	7920	7060	7100	13110	10080
	97.31	98.08	98.46	99.23	97.69	98.08	99.23	98.85	98.85	98.46	96.15*	97.31
200	10900	13120	10300	8660	10160	6120	10300	13000	10720	8860	7120	9620
	98.46	98.08	99.62	99.62	98.08	99.62	99.62	100	99.62	100	100	99.23
400	21640	25320	18120	16600	13480	12000	21160	30560	19400	14520	13360	14840
	99.62	99.23	99.23	99.62	100	99.62	100	99.62	100	100	100	99.62
	200						300					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	6535	8250	7055	7755	11690	10765	8005	14135	5675	10475	19090	22115
	98.46	97.69	99.23	98.08	96.92	96.92	98.46	96.92	99.62	97.69	96.15*	96.15*
100	16050	10580	6150	8120	15430	16560	8000	13790	11130	15530	21340	20720
	97.69	99.23	100	99.23	98.08	97.31	99.62	98.85	99.23	98.46	97.69	98.08
200	16640	12840	16640	8900	14200	17200	14680	20380	8820	14260	13160	6560
	99.23	100	99.23	100	99.23	98.85	99.62	99.62	100	99.62	99.62	100
400	21080	30440	20840	16320	21720	17960	31960	27960	21400	17680	25320	22920
	100	100	100	100	99.62	99.62	99.62	100	100	100	99.62	99.62

Table 2: Results for Triangle1.

Within an EDA, at each generation, half of the population was selected according to a rank-based strategy. New individuals were simulated from the learnt probability distribution by means of Probabilistic Logic Sampling, and the population was created in an elitist way.

In Tables 2 to 8, the results from the experiments are shown. In each cell, the average results from the ten executions are provided. The first row provides the average number of generated test inputs during the process and the following is the average coverage measurement, if the highest coverage achieved in the ten executions is not 100% then this value is labelled with an asterisk.

5.2 EDAs performance

Results presented by Tables 2 to 8 reveal that in 94% of the cases the average coverage surpasses 90%, and if the population is big enough, 100% is reached for every programme. Although tables do not show this, when the highest achieved coverage is not 100% (asterisk values) then the best execution obtained a value greater than 90% in all cases with the exception of three from the Triangle4 programme, which reached more than 88%.

The hardest programmes for the test case generator seem to be Triangle4 and Atof. Table 1 reveals that Triangle4 has no compound expression in its conditional statements although the nesting depth is one of the largest. In fact, several of the expressions are of the form $\mathcal{A} = \mathcal{B}$, which are usually the most difficult to fulfil. With regard to the Atof programme, the nesting depth is the highest and three-quarters of the expressions are compound. The comparison operator in many of the subexpressions are also equalities.

In order to observe the behaviour of the objective selection phase in the generator, for each pro-

pop.size	max. generations											
	50						100					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	3315	4215	3225	4920	5145	4305	7365	7620	5565	7385	6215	8885
	96.92	96.54	98.08	97.69	95.77	96.15	96.92	97.31	98.46	96.15	97.69	98.85
100	7330	6330	5730	4420	9860	9020	9120	9980	10430	6980	13150	18120
	97.69	98.46	97.69	98.46	96.54	95.39	98.46	98.08	98.08	98.85	97.31	96.54
200	8840	9780	7360	8220	9660	8320	10940	6200	16300	19920	19980	8400
	98.85	98.85	99.23	99.23	97.69	98.85	99.23	100	98.85	97.69	97.69	99.23
400	12440	15720	16520	11680	11760	10920	211160	15160	13680	18160	16680	9360
	100	100	99.62	99.62	99.23	100	100	100	99.62	99.23	99.23	100
	200						300					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	4085	7960	8555	10225	13825	17230	149755	25495	12795	10715	11385	16770
	99.23	99.23	98.85	98.85	98.85	97.69	97.69	96.92	98.08	98.85	99.23	99.62
100	11080	14220	11510	16790	20360	21520	15470	21210	20600	17820	33980	23150
	98.46	98.46	98.85	98.46	98.46	98.08	99.23	98.08	98.85	99.23	98.85	98.46
200	14920	23280	17960	17500	20820	12960	29060	13820	35100	32800	49000	23940
	100	99.23	99.23	99.62	99.23	99.23	98.85	99.62	98.46	98.85	97.69	99.23
400	10640	14000	33400	12920	33960	26080	16280	12880	10800	21840	55320	22560
	100	100	99.85	100	99.23	99.62	100	100	100	99.62	98.85	99.62

Table 3: Results for Triangle2.

pop. size	max. generations											
	50						100					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	3150	2950	2960	3700	4485	4450	3090	3570	3780	4435	4990	7675
	98	98	99	96	94	94.5	99.5	98.5	98	97.5	97.5	95.5
100	5440	5780	5990	3820	6930	5640	4790	5780	6740	3880	7240	4730
	98.5	97.5	98	99.5	95	97.5	100	100	99	100	98	99
200	10500	9600	7820	8160	8300	8520	9300	11080	10340	8920	12940	10560
	99	97.5	100	99.5	98	99	100	100	99.5	100	98	100
400	17600	21960	18840	18520	15480	16240	19400	27200	22320	15960	15320	15320
	98.5	95.5	99.5	99.5	100	99.5	100	100	99.5	100	100	100
	200						300					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	4540	4700	5965	4965	9585	11500	3875	6750	4805	6580	15920	16405
	99	99	99	99	98	98.5	100	98.5	99.5	99.5	98.5	98
100	5990	6220	7460	5330	7930	8010	6320	9240	8990	4080	12650	6660
	100	100	99.5	100	99.5	99.5	100	99.5	99.5	100	99	100
200	10160	13960	10600	8600	13920	21380	10640	10420	8460	8640	13500	7080
	100	100	100	100	99	98.5	100	100	100	100	100	100
400	18680	25000	21280	15440	21000	16440	20800	29600	21080	15200	21080	15800
	100	100	100	100	100	100	100	100	100	100	100	100

Table 4: Results for Triangle3.

pop. size	max. generations											
	50						100					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	21975	20500	26205	10535	12630	12530	30700	26355	28955	18460	24300	24145
	88.85*	89.62*	88.85*	89.23*	88.85*	85.39*	89.23*	88.08*	90.39*	89.62*	86.15*	88.08*
100	47600	48870	45870	24520	21140	16970	55660	48190	51010	30980	36930	40630
	90.77*	93.08*	93.46*	93.46	90.39*	90.77*	91.54	96.15	93.85	94.23	90*	90*
200	69820	96400	94980	37280	23500	17700	106960	99340	81320	31020	50560	28060
	97.31	98.08	94.23	97.69	95.39	96.92	96.54	96.54	96.92	99.23	93.46	97.69
400	128120	119120	105400	43160	27960	22720	154000	158520	157480	45720	22760	48240
	96.54	96.92	97.31	99.62	99.23	100	98.08	98.08	98.46	100	100	98.46
	200						300					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	50030	49755	43105	35835	49445	42035	53630	57515	47260	47815	67930	72610
	89.62*	87.69*	91.15*	91.15*	86.92*	86.92*	93.46	91.15*	93.08*	90.77*	87.69*	86.92*
100	84000	77980	80290	51270	64230	68200	114520	113360	79920	55230	98600	118550
	93.08	95	91.92	93.08*	90.39*	90.39*	92.69	92.31*	93.46	94.23*	90.77*	89.23*
200	92220	107560	130800	43780	73680	67100	180200	151800	133480	52620	138280	132520
	97.69	98.46	95.39	98.46	93.85	95.77	96.15	97.31	96.92	98.46	93.85	93.85*
400	147000	122040	168960	60840	50200	34560	225800	174880	141920	60800	86080	24080
	99.23	99.62	98.08	99.23	98.85	100	100	98.85	99.23	100	98.46	100

Table 5: Results for Triangle4.

pop. size	max. generations											
	50						100					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	13330	12675	13525	14350	6600	6555	9640	34050	12795	32230	7685	8850
	87.33	87	89.33	87.33	97	98.33	98.33	81.33*	97	83.67	100	98
100	28850	17740	28310	30080	20380	13840	36670	37050	18090	40360	21970	20770
	87	94.67	86.67	87	95	97.67	94.67	91.67	100	92	97	98
200	73780	46180	63140	62760	59020	72000	139280	43400	92780	119920	50460	54820
	83	92.67	84.67	87	88.33	84.33	83	98.67	91.33	88.33	96.33	96
400	165400	164920	148760	157640	150720	131320	193720	125040	146160	150480	158040	196560
	80.67*	88	88	87.33	80.67*	91.33	94	97	92.67	95	93.67	94.33
	200						300					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	9905	41790	9125	15665	11850	11015	17125	60995	13350	34315	14190	9505
	100	88.67	99.67	99.33	99	99.67	99.33	88	100	96.67	99	100
100	31990	32340	22660	54370	28310	27800	32780	67940	30250	59550	27560	31380
	98.67	97.67	100	94	98.67	99	99.67	94*	100	99.33	99.33	99.33
200	79920	89280	51400	104020	74760	68140	66100	119500	74020	115880	86260	65880
	98.33	95	100	96.67	97	97.67	99.33	94.67	99	98	98.67	98.67
400	150520	105560	214480	259400	160160	147320	179920	180280	203200	203480	187520	221680
	98.67	99	96	95.33	98.33	98.67	98	98	100	98.67	98.67	97.33

Table 6: Results for Atof.

pop. size	max. generations											
	50						100					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	2525	3275	2560	2360	3235	2370	6870	5985	4570	4745	6010	5510
	95.56	94.44	95.56	96.11	93.89	96.67	92.78*	93.89	95.56	95.56	93.89	94.44
100	5310	6700	6310	4810	5130	5790	8410	14620	11690	7010	6260	7860
	95	93.89	93.89	95.56	95.56	94.44	96.11	92.22	93.89	96.67	97.22	96.11
200	14140	13360	11340	4240	4920	6000	20600	20180	25500	12140	15920	2360
	92.78	93.89	94.44	98.89	98.33	97.78	95	95	93.33	97.22	96.11	100
400	22520	28280	20640	10160	8200	5840	46640	43000	39000	15680	4200	11080
	95	93.89	95	98.33	98.89	99.44	93.89	94.44	95	98.33	100	98.89
	200						300					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	12460	9830	13350	10295	6700	10475	12715	18555	18410	12490	17025	16995
	93.33	95	92.78	95	96.67	94.44	95.56	93.89	93.33	95.56	94.44	93.89
100	15410	17370	20890	20730	24760	16870	48440	31430	22270	13110	30850	24850
	96.11	95.56	94.44	94.44	93.33	96.67	91.11*	94.44	96.11	97.78	95	95.56
200	26820	41920	26000	14360	18040	29540	73320	50860	49820	31820	25520	19740
	96.67	94.44	96.67	98.33	97.78	96.11	93.33	95.56	95.56	97.22	97.78	98.33
400	83760	62440	74600	19160	4000	3800	99640	136120	99240	15920	3920	4320
	94.44	96.11	95	98.89	100	100	95.56	93.89	95.56	99.44	100	100

Table 7: Results for Remainder.

pop. size	max. generations											
	50						100					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	8030	7145	7765	6885	6855	6710	10570	12085	9845	9515	8555	10140
	93.18	94.09	93.18*	94.55	92.73	92.27	95.46	95.46	96.82	95.46	96.36	95.46
100	12150	13570	13690	11490	10440	10320	14090	16070	15010	16510	11060	13120
	96.36	95	95.46	95	95	95.91	97.27	99.09	97.27	95.91	99.09	99.09
200	20600	19920	21500	18100	13640	14060	24780	20340	18000	21800	18800	18380
	97.73	96.82	95.91	97.73	97.27	97.73	96.82	98.64	99.09	97.73	98.18	98.64
400	24560	18680	24600	22200	23840	23360	27560	25240	27960	22960	22080	24240
	98.18	99.55	98.18	99.55	97.27	97.27	99.09	99.09	100	100	100	99.55
	200						300					
	UMDA	PBIL	MIMIC	TREE	BIC	K2	UMDA	PBIL	MIMIC	TREE	BIC	K2
50	19875	17815	15480	17425	9320	10390	12275	22855	18760	18540	15510	12915
	96.82	95.91	98.18	96.82	99.09	97.27	98.64	95	98.18	97.27	98.64	99.09
100	29050	20540	20070	16360	10860	11930	20920	22350	17130	21360	19400	17270
	98.18	97.73	98.64	98.18	99.55	100	99.55	98.64	100	100	99.09	100
200	26200	21340	30300	22740	20780	21720	22440	30300	20580	25820	26200	30340
	99.55	100	99.09	100	99.55	99.09	100	100	100	100	100	100
400	24040	25560	34360	39120	33880	31520	31320	42640	23960	27800	30400	33920
	100	100	100	100	100	100	100	99.55	100	100	100	100

Table 8: Results for Complexbranch.

gramme, the number of times the search of an objective is repeated was recorded during the executions. The average is zero or near zero for all the programmes except Tiangle4 and Atof; from these two, Triangle4 takes higher values, with 7 as the maximum, while in Atof, the highest is 1.8. In both cases, this value decreases as the maximum number of generations increases, in fact, in Triangle4, it is near zero in most of the cases for 300 generations and in Atof, the value is zero or near zero for 100, 200 and 300 generations. Thus, according to these results and considering the set of individuals associated to a given objective, it seems that the number of times an individual that improves the fitness of the set is found, increases with the programme complexity for branch coverage. Therefore, the number of optimization problems being solved during the process, and hence the number of generated inputs, becomes higher.

Regarding the optimization phase in the test case generator, Tables 2 to 8 also provide interesting information concerning the differences between EDAs. Considering full coverage a mandatory requirement, multivariate EDAs offer the best results as they create the lowest or second lowest number of inputs in all the programmes except in Triangle3. Taking the ratio between generated inputs and achieved coverage into account, the best values are shared by bivariate and multivariate EDAs, to be precise, $\text{EBNA}_{\text{K2+pen}}$ takes the best ratio in three of the programmes, while TREE and MIMIC in two of the programmes each. These best ratios belong to the cases of 50 individuals and 50 generations, with the exception of Remainder, which obtained the best with 200 individuals and 100 generations.

In general, the worst results of EBNAs come given by small populations, thus making univariate EDAs competitive. However, as population size grows, multivariate EDAs improve their average coverage and, when 100% is reached, the number of generated inputs is usually lower than in the rest of EDAs. This indicates that, when the required conditions are met, EBNAs obtain the optimum in less generations than the other alternatives. These results serve as a reinforcement of the suggestion made in Section 4.1 about the adequacy of multivariate EDAs for dealing with this problem.

In order to statistically validate these concepts, two rankings based on hypothesis tests were carried out, one over the number of generated inputs and the other over the coverage measurement. For each programme, and for each value of population size and maximum number of generations, EDAs were ranked in the way next explained. First, EDAs are ordered according to their average value in the result being considered, i.e. number of generated inputs (increasing order) or coverage (decreasing order); if a tie occurs, the involved EDAs are ordered by their variance. Then, following the obtained order, several Mann-Whitney tests are performed, each of which designates a rank value to an EDA. The first sample in the test is formed by the data from the i-th EDA in the order, and the second sample is the data from the $(i+1)$-th EDA. If the test finds significative differences at a confidence level of 0.05 then the rank value of the i-th EDA plus 1 is given to the $(i+1)$-th EDA, othewise this EDA is designated with the same rank value as the i-th EDA and the first sample of the next test is extended with the

Programme name	UMDA	PBIL	MIMIC	TREE	BIC	K2
Triangle1	32	39	27	**21**	30	25
Triangle2	**17**	18	19	20	20	**17**
Triangle3	19	27	20	**16**	24	19
Triangle4	32	31	31	**19**	20	20
Atof	20	24	18	26	**16**	**16**
Remainder	25	27	25	18	17	**16**
Complexbranch	19	20	19	17	**16**	**16**
Total	164	186	159	137	143	**129**

Table 9: Rank of EDAs with regard to the number of generated inputs.

Programme name	UMDA	PBIL	MIMIC	TREE	BIC	K2
Triangle1	**16**	17	**16**	**16**	18	18
Triangle2	**16**	**16**	**16**	17	17	**16**
Triangle3	**16**	17	**16**	**16**	20	**16**
Triangle4	20	19	20	**16**	26	26
Atof	22	24	18	22	19	**17**
Remainder	24	23	22	17	17	**16**
Complexbranch	17	18	**16**	18	**16**	**16**
Total	131	134	124	**122**	133	125

Table 10: Rank of EDAs with regard to the coverage measurement.

data from the $(i+1)$-th EDA. Once a rank is obtained for all the different combinations of population size and maximum number of generations, the sum of the rank values of each EDA is calculated.

Tables 9 and 10 show the ranks for the number of generated inputs and coverage measurement respectively. The last row of each table provides, for each EDA, the sum of the ranks in all the programmes. The best values are marked in bold. As can be observed, with regard to the number of inputs, multivariate EDAs take a larger number of the best values than other types of EDAs. Considering the last row the best EDA is $\text{EBNA}_{\text{K2+pen}}$, followed by TREE and EBNA_{BIC}. The PBIL algorithm is the worst EDA, although it must be noted that its results heavily depend on the α parameter, as was shown in (González et al., 2000). Taking the coverage measurement into account, the differences are not so clear. The total rank in the last row reveals that the values of the first three EDAs are quite similar and a difference of 12 units between the first and the last EDA. In this case,

TREE offers the lowest value, $\text{EBNA}_{\text{K2+pen}}$ is the second best EDA and MIMIC the third. Thus, it can be concluded that the best overall EDAs in the experiments, with regard to the coverage and generated inputs results, are TREE and $\text{EBNA}_{\text{K2+pen}}$.

5.3 Comparison with other works

All the works from which the programmes here considered were extracted use a GA. When comparing the results from Tables 2 to 8 with those of their respective works, it must be noted that the plain form of EDAs was applied in the experiments, while in the referred works, sophisticated forms of GAs are used. In (Wegener et al., 2001), a coarse grained parallel GA is chosen, in (Sthamer, 1996), different genetic operators and parameters are considered, and in (McGraw et al., 1998), a simple GA and a differential GA are employed. The fitness function also differs from the one in two of the works. In (Wegener et al., 2001), the fitness is calculated from a function of the type described in Section 2 and a measurement of the proximity to the conditional statement of the branch objective. In the case of (Sthamer, 1996), a function of the type in Section 2 is presented taking two distances for each programme into account, one is the absolute value of the difference between the numerical representations of the operands and the other is the Hamming distance between the binary representations.

However, the results here attained are better in most of the cases, therefore, EDAs seem to perform with great success as, regardless of the fitness function, constitute the major difference with respect to the previously refered works.

Apropos of (Wegener et al., 2001), 100% average coverage is achieved for all the programmes. Considering the input parameter intervals used here and the best results reaching 100% average coverage, in Triangle1 and Triangle2 near 64% and 85% less inputs respectively are generated, for Atof, the difference is 78% lower, and in the case of Complexbranch the decrease is 41%.

In (Sthamer, 1996), the presented parameter interval for the Triangle4 programme is $[-100, 100]$; fixing the coverage to 100% and considering the best results, approximately 12% less inputs are created here with regard to the results with the Hamming distance and nearly 18% less with respect to the distance based on the difference of absolute values. In the Remainder programme, the employed parameter interval in (Sthamer, 1996) was $[-20000, 20000]$; in this programme, the results are not outperformed, with 100% coverage fixed, the best results for the Hamming distance generate 72% less inputs than here and for the absolute value of differences based distance, the difference is 60% lower.

Results in (McGraw et al., 1998) are also improved, however, it must be taken into account that this work dealt with condition/decision coverage, which is a harder criterion involving branch coverage. The presented results in the referred work do not show the number of generated inputs, however, the best coverage reached in the five performed executions is 93%, while 100% average coverage is attained

here.

The results of these experiments conform to those in the application of EDAs to other problems (Inza et al., 2000): although in EDAs the quality of the solution is similar to the one achieved by GAs, the number of required generations and, hence, generated individuals is remarkably lower.

6 Conclusions

In this work a new approach for the automatic generation of test cases was described. In order to fulfil the branch coverage criterion, test inputs are searched by means of EDAs. Analysing the obtained results from the experiments, EDAs present as a powerful option for tackling this problem. The attained coverage was 100% and the number of generated inputs was significantly lower than in other works. Among the different EDAs, multivariate algorithms seem to be a promising alternative. Their capacity of expressing the dependencies among problem variables could be a key point, as these dependencies usually exist when trying the coverage of a particular branch.

At the present, the working line the authors are following is orientated towards the study of an approach that, during the test case generation, manages different lengths for the interval of values associated to an input parameter. This is an interesting question, as a small interval will decrease the search space size and could simplify the problem; on the other hand, if the interval is not appropriately chosen, an optimal solution could not even exist.

References

Bäck, T. 1996. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press.

Baluja, S. 1994. Population Based Incremental Learning: A method for Integrating Genetic Search Based Function Optimization and Competitive Learning. *Technical Report CMU-CS-94-163*. Carnegie Mellon University, Pittsburgh.

Baluja, S., and S. Davies. 1997. Combining multiple optimization with optimal dependency trees. *Technical Report CMU-CS-97-157*. Carnegie Mellon University, Pittsburgh.

Beizer, B. 1990. *Software Testing Techniques*. Van Nostrand Rheinhold.

Burton, S. 2000. Automated Testing from Z Specifications. *Technical Report YCS329*. Department of Computer Science, University of York, Heslington, York.

Castillo, E., J. Gutiérrez, and A. Hadi. 1997. *Expert Systems and Probabilistic Network Models*. Springer-Verlag.

Chow, C., and C. Liu. 1968. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on information Theory* 14:462-467.

Cooper, G. F., and E. A. Herskovits. 1992. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning* 9:309-347.

Corno F., P. Prinetto, M. Rebaudengo, and M. Sonza-Reorda. 1996. Exploiting Competing Subpopulations for Automatic Generation of Test Sequences for Digital Circuits. In *Proceedings of the Fourth International Conference on Parallel Problem Solving from Nature*, 792-800. Springer.

De Bonet, J. S., C. L. Isbell, and P. Viola. 1997. MIMIC: Finding Optima by Estimating Probability Densities. In *Advances in Neural Information Processing Systems 9*, 424-430. MIT Press.

Demillo, R., and A. Offut. 1993. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology* 2(2):109-127.

Duran, J. W., and S. C. Ntafos. 1984. An evaluation of random testing. *IEEE Transactions on Software Engineering* 10:438-444.

Fenton, N. E. 1985. The structural complexity of flowgraphs. In *Graph Theory with Applications to Algorithms and Computer Science*, ed.s Y. Alavi, G. Chartrand, L. Lesniak, D. R. Lick, C. E. Wall, 273-282. John Wiley & Sons.

Glover, F., and M. Laguna. 1997. *Tabu search*. Kluwer Academic Publishers.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

González, C., J. A. Lozano, and P. Larrañaga. 2000. Analyzing the Population Based Incremental Learning Algorithm by Means of Discrete Dynamical Systems. *Complex Systems* 12(4):465-479.

Grefenstette, J. J. 1986. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* 16(1):122-128.

Henrion, M. 1988. Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In *Proceedings of the Second Conference on Uncertainty in Artificial Intelligence*, 149-163. North-Holland.

I. Inza, P. Larrañaga, R. Etxeberria, and B. Sierra. 2000. Feature Subset Selection by Bayesian networks based optimization. *Artificial Intelligence* 123(1-2):157-184.

Jones, B., H. Sthamer, and D. Eyres. 1996. Automatic structural testing using genetic algorithms. *Software Engineering Journal* 11(5):299-306.

Kirkpatrick, S., C. D. Gellat, and M. P. Vecchi. 1983. Optimization by simulated annealing. *Science* 220:671-680.

Korel, B. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering* 16(8):870-879.

Larrañaga, P., R. Etxeberria, J. A. Lozano, and J. M. Peña. 2000. Combinatorial optimization by learning and simulation of Bayesian networks. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, 343-352. Morgan Kaufmann.

Larrañaga, P., and J. A. Lozano. 2002. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation*. Kluwer Academic Publishers.

McGraw, G., C. Michael, and M. Schatz. 1998. Generating software test data by evolution. *Technical Report RSTR-018-97-01*. RST Corporation, Sterling, Virginia, USA.

Mühlenbein, H., and G. Paaß. 1996. From recombination of genes to the estimation of distributions I. Binary Parameters. In *Proceedings of the Fourth International Conference on Parallel Problem Solving from Nature*, 178-187. Springer.

Mühlenbein, H. 1998. The equation for response to selection and its use for prediction. *Evolutionary Computation* 5(3):303-346.

Ould, M. 1991. Testing- a challenge to method and tool developers. *Software Engineering Journal* 6(2):59-64.

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

Pelikan, M., D. E. Goldberg, and F. Lobo. 1999. A survey of optimization by bulding and using probabilistic models. *Technical Report IlliGAL Report 99018*. University of Illinois at Urbana-Champaing, Urbana.

Pohlheim, H., and J. Wegener. 1999. Testing the Temporal Behavior of Real-Time Software Modules using Extended Evolutionary Algorithms. In *Proceedings of the Second Genetic and Evolutionary Computation Conference*, 1795-1802. Morgan Kaufmann.

Roper, M., I. Maclean, A. Brooks, J. Miller, and M. Wood. 1995. Genetic Algorithms and the Automatic Generation of Test Data. *Technical Report EFoCS-19-95*. University of Stratchclyde, Glasgow.

Schwarz, G. 1978. Estimating the dimension of a model. *Annals of Statistics* 7(2):461-464.

Shepperd, M. 1993. *Software Engineering Metrics I: Measures and Validations*. McGraw-Hill.

Smith, J., and T. C. Fogarty. 1996. Evolving Software Test Data - GA's learn Self Expression. In *Proceedings of Evolutionary Computing. AISB Workshop*, ed. T. C. Fogarty, 346-354. Springer-Verlag.

Sthamer, H. 1996. The automatic generation of software test data using genetic algorithms. PhD Thesis. University of Glamorgan, Pontyprid, Wales, Great Britain.

Tracey, N., J. Clark, K. Mander, and J. McDermid. 1998. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, 285-288. IEEE CS Press.

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