Computational Intelligence, Volume 24, Number 1, 2008

DYNAMIC SEARCH SPACE TRANSFORMATIONS FOR SOFTWARE TEST DATA GENERATION

RAMÓN SAGARNA AND JOSÉ A. LOZANO

Intelligent Systems Group, Department of Computer Science and Artificial Intelligence, University of the Basque Country, Spain

Among the tasks in software testing, test data generation is particularly difficult and costly. In recent years, several approaches that use metaheuristic search techniques to automatically obtain the test inputs have been proposed. Although work in this field is very active, little attention has been paid to the selection of an appropriate search space. The present work describes an alternative to this issue. More precisely, two approaches which employ an Estimation of Distribution Algorithm as the metaheuristic technique are explained. In both cases, different regions are considered in the search for the test inputs. Moreover, to depart from a region near to the one containing the optimum, the definition of the initial search space incorporates static information extracted from the source code of the software under test. If this information is not enough to complete the definition, then a grid search method is used. According to the results of the experiments conducted, it is concluded that this is a promising option that can be used to enhance the test data generation process.

Key words: software testing, evolutionary algorithms, dynamic representations, estimation of distribution algorithms, search based test data generation.

1. INTRODUCTION

Testing is the means used in practice to verify the correctness of software produced. Considering the crucial role of software nowadays, it is not difficult to imagine the significance of testing. In fact, this phase from the software's life cycle usually accounts for 50% of project resources (Beizer 1990; Sommerville 2001). A huge amount of these resources is dedicated to the generation of the input cases to be applied to the program tested. This task is not trivial, as input cases must confom to the test type and its requirements. Because most organizations perform this step manually, the automatic generation of test data is worthwhile and has turned into one of the most challenging problems in the area.

Test case generation approaches may be classified as either statistical, functional or structural testing methods. Statistical testing samples the input domain according to a probability distribution that is obtained from the program's operational profile. Due to its simplicity, a common approach is to automatically generate test cases simulating a uniform distribution (Duran and Ntafos 1984). On the other hand, functional testing is based on the program specification. More precisely, inputs are generated taking the functional properties of the program into account, and its automatization demands a formal specification. As an example, in Burton (2000), a framework oriented to a popular specification language named Z was described.

Structural testing is based on the internal structure of the program. The source code reveals control or data flow entities such as the branches that the flow of control can take from a conditional statement or the different possible usages of a variable. According to these entities, several adequacy criteria are defined. For instance, branch coverage is a classical criterion stating that every program branch must be exercised. Other typical criteria are passing through every code statement (statement coverage), exercising every sequence of branches from the program input to its output (path coverage), or covering all the definition-usage pairs for each variable (all-defs coverage). Thus, a structural test data generator tries

Address correspondence to Ramón Sagarna, Paseo Manuel de Lardizabal 1, 20018 San Sebastián, Guipúzcoa, Spain; e-mail: ccbsaalr@si.ehu.es

to fulfill an adequacy criterion by producing the appropriate inputs. To know the level of completion attained by the generator, a coverage measurement indicates the percentage of entities exercised by a set of test cases for a given criterion. It should be marked that, in practice, coverage measurements and structural criteria may also be used by methods from other testing strategies and vice versa.

The two most common strategies to deal with the automation of structural testing are static and dynamic test data generation.

The main feature of the static strategy is that program execution is not required, because test cases are obtained through a static analysis of the source code. Most of the initial approaches were based on a technique named symbolic execution. This technique consists of choosing an entity from the program structure, and assigning a system of inequalities in terms of the input parameters. The system is built by substituting variables affecting the entity with symbolic values while respecting the constraints associated with 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, dynamic approaches execute the program to generate the test inputs. More precisely, an instrumented version of the program is constructed, that is, the source code is expanded with probe instructions that will extract information concerning the execution of an input. The information collected is used to guide the search for the test case exercising the desired entity. In Korel (1990), the obtained information determined a function value assigned to each input. The objective was to find an input minimizing its function value, which only occurred when reaching the target entity.

In recent years, a number of approaches under the name of Search Based Software Test Data Generation (SBSTDG) has been developed, offering interesting results (McMinn 2004). The aim of SBSTDG is to seek test cases employing metaheuristic search techniques during the process. In other words, during the test data generation, an optimization problem is formulated. This problem is solved through the above mentioned metaheuristic algorithms.

It is common to select an objective entity and search for an input covering it. Obviously, this search is then carried out by means of a metaheuristic, e.g., Genetic Algorithms (McGraw, Michael, and Schatz 2001). In consequence, the domain of each input parameter becomes an important matter, because it defines the space where the search is performed. In fact, the space defined by the inputs and the objective function is often large and complex, making the coverage of an entity a difficult task. Most of the SBSTDG works dealing with this issue to date have concentrated on the optimization technique and the objective function. However, little attention has been paid to the selection of an appropriate search space. This is an interesting question, as focusing the search on a promising region could simplify the problem. On the other hand, if an adequate region is not chosen, an optimal solution (an input covering the entity) may not even exist.

In the context of Evolutionary Computation, this matter can be tackled by Self-Adaptive Representation methods. These methods may be classified as a form of parameter control (Eiben, Hinterding, and Michalewicz 1999) that, according to the behavior of the execution, dynamically transforms an individual's representation and, thus, the search space. Although it depends on the method, generally, the purpose of the transformation is to direct the search toward the most promising region found so far and avoid getting stuck in local optima (Whitley, Mathias, and Fitzforn 1991).

The present work describes an alternative to the search space selection issue in test data generation. The two major concepts which support this alternative are the use of a priori knowledge on the problem instance to choose a search region, and modifying this region through the solution's representation. More precisely, these concepts are applied to a SBSTDG approach for branch coverage of C/C++ programs. Initially, the metaheuristic seeks for in

a region chosen from the whole feasible search space. To select a promising region, its definition is based on static information extracted from the program's source code. In case this information is not useful to the definition, then a grid search method is applied. Additionally, during the process, the size of the region is increasingly widened. This way, if the objective entity is not exercised, a new search is performed on a larger region. This enlargement is applied to the approach from two points of view, giving rise to two algorithms. In both algorithms, the metaheuristic technique employed to deal with the optimization problem is an Estimation of Distribution Algorithm (EDA) (Larrañaga and Lozano 2002; Lozano et al. 2006).

EDAs are a set of Evolutionary Algorithms which, instead of creating new individuals through the classical recombination operators, estimate the probability distribution associated with the selected individuals and sample this distribution to create the next population. This technique has already been applied to the test data generation problem with great success (Sagarna and Lozano 2005). Therefore, it constitutes an adequate benchmark for comparison with the proposed option.

Several experiments were conducted to know whether the algorithms exposed here may help to enhance the test case generation process. Performance of both alternatives is evaluated and compared with each other. Moreover, the results face those obtained with EDAs, and interesting conclusions are obtained about the effect of search space size on the behavior of the test data generator.

The remaining sections are organized as follows. First, EDAs are briefly introduced, though a little more attention is paid to the one used in the experiments: the TREE algorithm. After describing SBSTDG and a few salient references from this field, the alternative developed in this work is explained. In the next section, the experiments and the analysis of their results are shown. Finally, a short summary of the work and conclusions obtained are included.

2. INTRODUCTION TO ESTIMATION OF DISTRIBUTION ALGORITHMS

The term Estimation of Distribution Algorithm (EDA) (Mühlenbein and Paaß 1996; Larrañaga and Lozano 2002; Pelikan, Goldberg, and Lobo 2002) alludes to a family of Evolutionary Algorithms which represents an alternative to the classical optimization methods in the area. Algorithmically, a Genetic Algorithm and an EDA only differ in the procedure to generate new individuals. Instead of using the typical breeding operators, EDAs perform this task by sampling a probability distribution previously built from the set of selected individuals. Indeed, this distribution is responsible for one of the main characteristics of these algorithms, that is, the explicit description of the relationships between the problem variables.

2.1. Abstract EDA

To simplify the discussion, only discrete domains are considered in the following. For a more extended description, including continuous domains, refer to Larrañaga and Lozano (2002).

Given an n-dimensional random variable $\boldsymbol{X} = (X_1, X_2, \ldots, X_n$ and a possible instantiation $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, the joint probability distribution of $\boldsymbol{X}$ will be 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 $X_j = x_j$ will be represented as $p(x_i \mid x_j) = p(X_i = x_i \mid X_j = x_j)$. In the context of Evolutionary Algorithms, an individual of length n can be considered an instantiation $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ of $\boldsymbol{X} = (X_1, X_2, \ldots, X_n)$.

$D_0 \leftarrow$ Generate M individuals (the initial population) randomly

Repeat for $l = 1, 2, ...,$ until stopping criterion is 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 individuals selected

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

FIGURE 1. Abstract EDA pseudocode.

Let the population of the lth generation be D_l. The individuals selected D_l^{Se} constitute a data set of N cases of $\boldsymbol{X} = (X_1, X_2, \ldots, X_n)$. EDAs estimate $p(\boldsymbol{x})$ from D_l^{Se}, therefore, the joint probability distribution of the lth generation will be represented by $p_l(\boldsymbol{x}) = p(\boldsymbol{x} \mid D_{l-1}^{Se})$. New individuals are then obtained sampling $p_l(\boldsymbol{x})$. A pseudocode for the abstract EDA is presented in Figure 1.

The key point of EDAs is how the probability distribution is estimated at each generation. The computation of all the parameters of $p_l(\boldsymbol{x})$ is unviable because they are, at least, $2^n - 1$ (the case of binary variables). Thus, it is factorized according to a probability model that, in some cases, limits the possible dependencies among the variables $X_1, X_2, \ldots, X_n$. This leads to several approximations assuming different levels of complexity in their models. The alternatives range from those where the variables are mutually independent to those with no restrictions on variable interdependencies. The most restrictive models avoid the induction of probability distributions with dependencies. However, they allow for a fast and easy estimation that may convert them into a suitable possibility to solve a problem. On the other hand, the least restrictive models are able to show all the dependencies between the variables in a problem even though their computational cost is expensive and, in some cases, can result in an impractical choice.

2.2. EDA Instances

The problem of model induction has been tackled by separate scientific fields, such as statistical physics or probabilistic reasoning. EDAs benefit from this knowledge through interdisciplinary research. Taking probabilistic model complexity into account, they may be classified as univariate, bivariate and multivariate.

Univariate EDAs assume that the n-dimensional joint probability distribution is decomposed as a product of n univariate probability distributions, that is:

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

For example, the Univariate Marginal Distribution Algorithm (Mühlenbein 1998) is an instance belonging to this category, which estimates $p_l(x_i)$ as the relative frequencies of x_i in data set D_{l-1}^{Se}.

A weakness of this kind of EDAs arises if dependencies between the problem variables exist because, obviously, the factorization of $p_l(\boldsymbol{x})$ cannot represent them. To a certain extent, this drawback is overcomed by bivariate EDAs. These approximations make use of second

order statistics to estimate the probability distribution. Hence, apart from the probability values, a structure that reflects the dependencies among the variables must be given. The factorization carried out by the models in this category can be expressed as follows:

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

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

Bivariate EDAs restrict to order one dependencies. However, with the models from multivariate EDAs, it is possible to express all the variable interdependencies existing in a problem. In this last category, the probability distribution is estimated by means of probabilistic graphical models (Castillo, Gutiérrez, and Hadi 1997), which use a graph to represent the detected dependencies between the variables. Thus, the factorization associated with this type of EDAs is as follows:

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

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

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 learned from D_{l-1}^{Se}. A Bayesian network 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 S leads to different EBNA instantiations, e.g., EBNA_{BIC}.

2.2.1. TREE. A bivariate EDA is TREE (Larrañaga and Lozano 2002), the algorithm used for the experiments conducted to evaluate the approach in this paper.

TREE refers to an adaptation of the Combining Optimizers with Mutual Information Trees (COMIT) algorithm (Baluja and Davics 1997).

In COMIT, $p_l(\boldsymbol{x})$ is estimated through the Maximum Weight Spanning Tree algorithm (Chow and Liu 1968). The objective of Chow and Liu was to find the first order dependence probability distribution $p^t(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i \mid x_{j(i)})$ which best approximates $p(\boldsymbol{x})$. As a discrepancy measure, the Kullback-Leibler cross-entropy from p^t to p ($KL(p \mid p^t)$) was chosen (Kullback and Leibler 1951). More precisely, the authors decomposed $KL(p \mid p^t)$ as follows:

$$KL(p \mid p^t) = -\sum_{i=1}^{n} I(X_i, X_{j(i)}) + \sum_{i=1}^{n} H(X_i) - H(\boldsymbol{X}),$$

where $I(X_i, X_{j(i)})$ is the mutual information measure between X_i and $X_{j(i)}$, and $H(X_i)$ and $H(\boldsymbol{X})$ denote the entropy of p with regard to X_i and $\boldsymbol{X}$, respectively. Because the values of $H(\boldsymbol{X})$ and $H(X_i)$ for all i are not influenced by the dependencies in p^t, minimizing $KL(p \mid p^t)$ is equivalent to maximizing $\sum_{i=1}^{n} I(X_i, X_{j(i)})$. To obtain the best approximation, the algorithm considers the set of tree structures where each edge is weighted with the mutual information between the variables involved. Then, they propose a simple method to obtain the structure with the maximum sum of weights, which is the structure of the p^t minimizing $KL(p \mid p^t)$.

Once an estimation of $p_l(\boldsymbol{x})$ is obtained, COMIT samples a number of individuals from $p_l(\boldsymbol{x})$ and selects the best as the initial solutions of a local search. The resulting individuals are then used to create a new population. In TREE, this local search step is eliminated and, thus, the next population is obtained directly from $p_l(\boldsymbol{x})$.

3. SEARCH BASED SOFTWARE TEST DATA GENERATION

Search Based Software Test Data Generation (SBSTDG) is an emerging field which refers to the selection of software test cases making use of metaheuristic search techniques.

A number of approaches have been already proposed for different testing types, e.g., functional testing (Tracey 2000). However, the present work deals with branch coverage, thus these other approaches are out of the scope of this paper and, in the following, only structural testing is discussed. A well-crafted and extensive review of SBSTDG can be consulted in McMinn (2004).

3.1. The General Scheme

Most of the works developed for structural testing to date are based on a dynamic test data generation strategy. Moreover, a huge amount of these works consist of choosing the entities to be exercised and, then, searching for the inputs covering them via a metaheuristic. Thus, it is common to more or less follow the general scheme in Figure 2. This scheme is an iterative two-step process where, firstly, a previously identified structural entity is selected (a branch, for instance) and marked as an objective. In the second step, the objective entity is assigned a function dependent on the program input, and its optimization is sought. This objective function is formulated in such a way that, if an executed input exercises the objective, the value is optimum. Otherwise, the value is proportional to how close the input is to the objective coverage. Consequently, to obtain the function value of an input it must be previously executed on an instrumented version of the program which will provide the information necessary.

This way, the test case generation is tackled as the resolution of a number of function optimization problems, one for each objective entity.

Despite the different selection options, the most common practice is to determine the objective entity with the help of a graph that reflects the structural characteristics of the program. In the case of branch coverage, a control flow graph (Fenton 1985) is typically employed. In such a graph, each vertex represents a basic block in the code, i.e., a maximal sequence of code statements such that if one is executed, then all of them are. There is an arc (x, y) if the control of the program can be transferred from block x to y without crossing any other block. Hence, in this kind of graph, a program branch is defined by every vertex x with $outdegree(x) > 1$.

The next step of the scheme in Figure 2 tackles an optimization problem. Formally, given the search space Ω formed by the program inputs and a function $h : \Omega \longrightarrow \text{IR}$, find $\boldsymbol{x}^* \in \Omega$ such that $h(\boldsymbol{x}^*) \leq h(\boldsymbol{x})\ \forall \boldsymbol{x} \in \Omega$.

For branch coverage, a classical strategy to create the objective function is the following. Given an objective branch b and an expression $\mathcal{A}\ \mathbf{OP}\ \mathcal{B}$ of the conditional statement **COND** associated with b in the code, with **OP** denoting a comparison operator, the value for an input $\boldsymbol{x}$ is determined by:

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

FIGURE 2. General scheme for test case generation.

$$h(\boldsymbol{x}) = \begin{cases} M & \text{if } \textbf{COND} \text{ not reached} \\ d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}}) + K & \text{if } \textbf{COND} \text{ reached and } b \text{ not attained} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where M is the largest computable value, $\mathcal{A}_{\boldsymbol{x}}$ and $\mathcal{B}_{\boldsymbol{x}}$ are appropriate representations of the values taken by $\mathcal{A}$ and $\mathcal{B}$ in the execution, d is a distance measurement, and $K > 0$ is a previously defined constant. Typically, 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 case **COND** involves 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 applying the associative and commutative properties to different logical expressions, the overall value for h is obtained.

3.2. Improving the Objective Function

The previous type of objective function suffers from well-known drawbacks, some of which have no clear solution yet. For example, if the comparison operator in the conditional expression is $\neq$, the function only takes three values and becomes plateau shaped. To solve this flaw, several possibilities based on code transformations are described in Harman et al. (2002b) and Baresel and Sthamer (2003). In Bottaci (2003), other weaknesses are identified and a number of alternatives are proposed to overcome them.

To a certain extent, these limitations may be alleviated with the objective function presented in Wegener, Baresel, and Sthamer (2001). In addition to the distance in the conditional statement **COND** of the objective branch, a *condition distance* is used for the inputs not reaching **COND**. This distance considers the path in the control flow graph taken by an input during execution. The value is calculated in terms of the number of branching vertices straying from the subpath between **COND** and its nearest vertex in the execution path. Therefore, the function in equation (1) is extended, maintaining the notation, as follows:

$$h(\boldsymbol{x}) = \begin{cases} d_c(v_c, v_n) & \text{if } \textbf{COND} \text{ not reached} \\ \frac{d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}}) + K}{L + (d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}}) + K)} & \text{if } \textbf{COND} \text{ reached and } b \text{ not attained,} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where v_c is the vertex in the control flow graph representing **COND**, v_n is the nearest vertex to **COND** in the path followed by $\boldsymbol{x}$, d_c is the condition distance, and $L > 0$ is a previously defined constant. Notice that L is employed to ensure that the function value when **COND** is not reached surpasses the value when **COND** is reached but b is not attained.

In this manner, if an input was unable to reach the condition, instead of assigning it the worst value (M), the proximity to the condition is taken into account and the objective function is smoothed with regard to equation (1).

3.3. Applied Metaheuristics and Extensions

Apropos the metaheuristic employed to solve the optimization problem, the most prevalent choice has been the Genetic Algorithm. This technique was applied for branch coverage

in Sthamer (1996) and Wegener et al. (2001). The former compared binary and gray coded representations for the program inputs. However, no clear conclusion could be drawn as to which of them was superior. In the latter work, excellent coverage results where obtained with a parallel Genetic Algorithm using a function of the form of equation (2) to calculate the fitness of the individuals. In contrast, in the work by Pargas, Harrold, and Peck (1999), fitness is only the condition distance described above. Genetic Algorithms have also been chosen for other testing criteria like, for instance, path coverage (Lin and Yeh 2001) and condition/decision coverage (McGraw et al. 2001). Metaheuristics proposed in other works include Simulated Annealing (Tracey et al. 1998), Tabu Search (Díaz, Tuya, and Blanco 2003) and, more recently, EDAs (Sagarna and Lozano 2005); all tackling branch coverage with the classical objective function. In Sagarna and Lozano (2006), an emerging method known as Scatter Search was selected for the optimization step. Besides, a collaborative scheme between this method and EDAs was developed, offering promising results.

Although the metaheuristic technique deals with one optimization problem at a time, the real goal of test case generation is to solve a set of problems. Several approaches in the literature have taken this into consideration to improve the process. The alternative suggested by some works is to profit from the good solutions found by not only evaluating an input for the current objective entity, but also with regard to all the others. Each entity is assigned a set containing the best inputs so far which are used to seed the initial phase of the metaheuristic (Wegener et al. 2001; Sagarna and Lozano 2005). Similarly, in McGraw et al. (2001), the set of an entity is composed of the inputs just reaching the condition associated with the entity. Moreover, this type of strategy is employed for different testing criteria. For instance, the work by Bueno and Jino (2002) deals with path coverage, and a set of inputs exercising a selected path is sought at each step; thus, the initial population of a Genetic Algorithm is seeded with the closest sets of inputs to covering the path from those stored in a base pool. In contrast, in the approach for path coverage described in Hermadi and Ahmed (2003), a multiobjective optimization view is adopted. This system uses a Genetic Algorithm where an individual represents an input and the fitness value is obtained from a weighted sum of the proximities to the coverage of each path.

Indeed, it should be marked that there are other strategies for structural test data generation, aside from the one outlined in Figure 2. For example, in Smith and Fogarty (1996), a Genetic Algorithm is used once again. However, 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 fulfill an adequacy criterion is faced from a pure Evolutionary Computation view, where an individual represents a solution to the whole problem.

3.4. An Example of the General Scheme

To sum up, the preprocessing required to automate the generation of test data for branch coverage following the general scheme in Figure 2 should be noticed. Figure 3 illustrates this by showing an example program and the elements to be induced from it: the control flow graph and the instrumented program version. The reduced box on the right represents the information supplied by a hypothetical execution of the instrumented program.

The graph is used to select the next objective branch whose coverage will be pursued, for example, branch (2, 3). A Genetic Algorithm could be used in the optimization phase. Thus, an individual is a representation of the program input, i.e., three integers. If the inputs set strategy described above is applied, the initial population of the Genetic Algorithm could be seeded with the set associated to branch (2, 3).

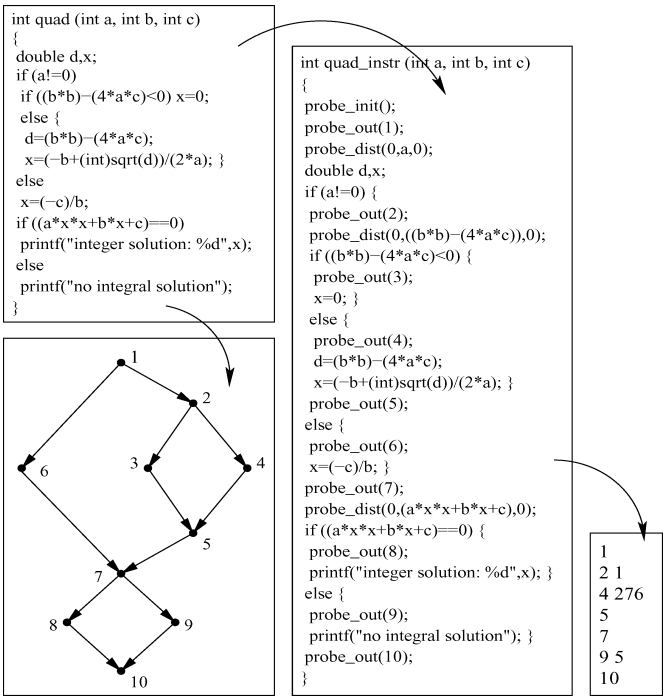

FIGURE 3. Example of source code, control flow graph, instrumented version, and output information.

Each input generated during the search is executed on the instrumented program version to elicit its objective function value. The instrumentation results shown in the reduced box of Figure 3 correspond to input (1, 20, 31). Each line of the box contains the traversed basic block (numbered as in the graph) and, if the previous block had a condition with an expression $\mathcal{A}\,\mathbf{OP}\,\mathcal{B}$, the value of $|\mathcal{A}-\mathcal{B}|$ in the execution. Using this information, the value of the condition distance (d_c) shown in equation (2) can be obtained. However, this is not necessary, as input (1, 20, 31) reaches the condition of branch (2, 3). Hence, according to equation (2) and taking $K = 1$ and $L = 1000$, $h(1, 20, 31) = \frac{276+1}{1000+276+1} = 0.2169$. Although the input is already evaluated for the Genetic Algorithm, the instrumentation results are used to calculate $h(1, 20, 31)$ with regard to the rest of the branches. This way, if (1, 20, 31) is a high-quality input, it is stored in the set of the corresponding branch.

Once the search finishes, a new round of the scheme in Figure 2 is performed until, for instance, every branch has been selected as an objective.

4. THE DYNAMIC SEARCH SPACE APPROACH TO SOFTWARE TEST DATA GENERATION

If a test data generation system deals with the problem as the process of solving a set of function optimizations, the search space becomes an important element. The present work describes an alternative which takes this into consideration in the context of a SBSTDG approach for branch coverage of C/C++ programs. More precisely, the region where the metaheuristic seeks for is initially defined with heuristic information obtained from the program's source code. During the process, the size of the region is increasingly widened so

that, if the optimum was not found in the current space, a new search is performed in a larger one. Two ways of tackling this region expansion are proposed, giving rise to two algorithms.

4.1. Motivation

In a SBSTDG approach following the scheme in Figure 2, the coverage of a branch may result in a highly difficult task, as the space defined by the inputs and the objective function is usually large and complex.

Most of the efforts to address this issue have concentrated on the objective function and the optimization technique. Attempts on the former relate to the concepts in Section 3.2, while, on the other hand, a clear case concerning the techniques is the general confrontation of local with global optimization procedures.

Surprisingly, so far little attention has been paid to the selection of an appropriate search space. This is an interesting matter, as focusing the search on a promising region could simplify the problem, while making an inadequate choice an optimal solution may not even exist.

An alternative facing this question is suggested in Harman et al. (2002a). Here, a dependence analysis is applied to the variables in the source code to identify the input parameters that cannot affect the coverage of a given branch. This way, a number of problem variables can be eliminated and the search space, reduced.

The present paper describes two approaches for the search space selection issue that follow the same line. Both extend the method explained in Sagarna and Lozano (2005a), so this work is outlined next.

4.2. Basic Approach

The system developed by Sagarna and Lozano conforms to the general scheme in Figure 2. Each code branch is associated with three possible states: covered, treated but uncovered, or untreated. Initially, all the branches are in the untreated state. After tackling the optimization problem of a branch, if the optimum was reached, the branch is marked as covered. Otherwise, its state is marked as treated but uncovered.

The stopping criterion of the scheme is full coverage achievement (all branches in the covered state) or unsuccessful treatment of every unexercised objective branch (treated but uncovered state).

Additionally, the input sets strategy discussed in Section 3.3 is applied. That is, for each branch, a set with the best inputs found so far is kept at every moment during the process. The quality of an input set is taken as the average objective function value of the elements in the set. Thus, the objective selection step consists of choosing the branch with the highest quality set.

The optimization step seeks inputs covering the objective branch through an EDA. An individual in the EDA is a bit string representing an input and the initial population is composed of the inputs in the set of the branch.

Figure 4 illustrates a schema of the whole process. At each iteration, a branch, together with its set of best inputs, is selected as the objective, and its coverage is sought through an EDA.

In the EDA, each individual (input) is evaluated not only for the objective branch, but with regard to every other uncovered branch. This way, if the individual improves the quality of the set of the branch, then the worst input in the set is replaced by the one represented by the individual. In this case, the quality of the set has been increased, so it may result in

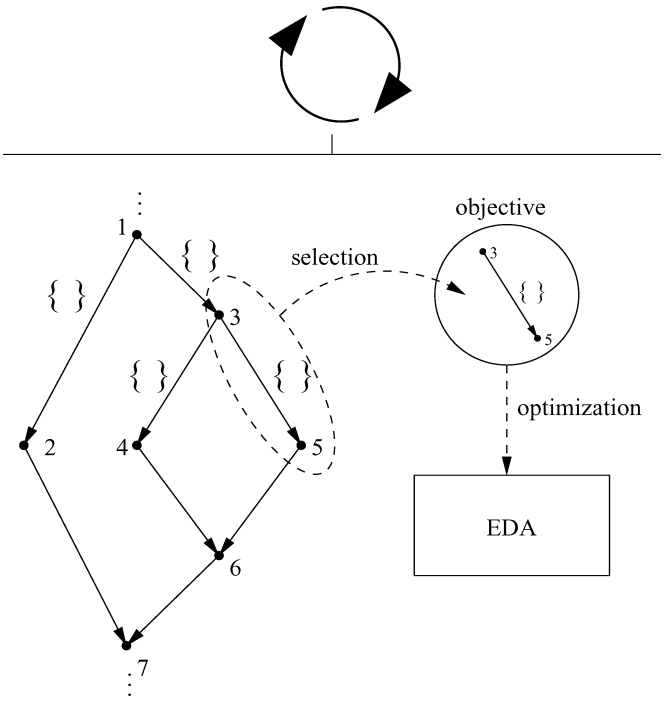

FIGURE 4. Schema of the basic approach.

a promising population seed. Hence, if the branch had previously been treated, its state is marked as untreated and becomes a new candidate objective in the selection step.

4.3. The Self-Adaptive Approach

The Self-Adaptive alternative to test data generation exposed here consists of selecting an initial search space and modifying its size for each uncovered branch. The space of an objective branch is defined by the interval of values that each input parameter of a program can take. To be precise, for each branch and each parameter, a value is chosen to be the center of the interval, and a maximum increment over the center defines the amplitude. The process departs from a small range of values for each parameter and, as branches remain uncovered, the range is increasingly augmented. Centers of the intervals are fixed for the whole process, thus, to start seeking on a promising region, static heuristic information from the program is used to locate these points. In case this information is not useful to identify a center, a grid search method is applied.

Two approaches following this line have been developed. One of them adapts the size of the search space for all the uncovered branches at a time. In the other approach, each region enlargement involves a single objective branch.

Multiple Objective Adaptation (MOA). The idea behind this method can be clearly stated: to use the general scheme in Figure 2 over widening regions. This leads to the left-side algorithm in Figure 5. Therefore, the basic approach is applied initially with a reduced interval of values for an input parameter and, once it is finished, if uncovered branches exist,

(1) Assign initial search region to each branch
(2) Repeat until stopping criterion is met
(3) Repeat until stopping criterion is met
(4) $O \leftarrow$ Select objective branch
(5) Apply EDA to cover O
(6) Enlarge region

(1) Assign initial search region to each branch
(2) Repeat until stopping criterion is met
(3) $O \leftarrow$ Select objective branch
(4) Repeat until stopping criterion is met
(5) Apply EDA to cover O
(6) Enlarge region

FIGURE 5. Algorithms for the MOA (left-hand panel) and SOA (right-hand panel) approaches.

it is applied again with a larger interval. The left-hand side of Figure 6 depicts an illustration of this idea.

Single Objective Adaptation (SOA). This alternative is similar to the basic approach except for the optimization step. Starting from a small search space, the EDA executes several times over increasingly augmented regions while the coverage of the objective branch is not attained. The right-hand side of Figures 5 and 6 represent the algorithm associated with this method and a schema of the process, respectively.

In the next pages, these two approaches are discussed in detail by first explaining the steps of their algorithms, and later, how the set of inputs is managed.

4.3.1. Algorithm Steps Description. The description applies to both MOA and SOA, because the same steps for each algorithm implement the same concepts.

4.3.1.1. Region Initialization—Step 1 (MOA, SOA). Each branch is assigned an initial search region which will have the smallest size. A reduced region allows for a fast search, although the chances of containing the global optimum may be few. Hence, to reach a high degree of efficiency, it is important to obtain an initial region that is near the optimal input.

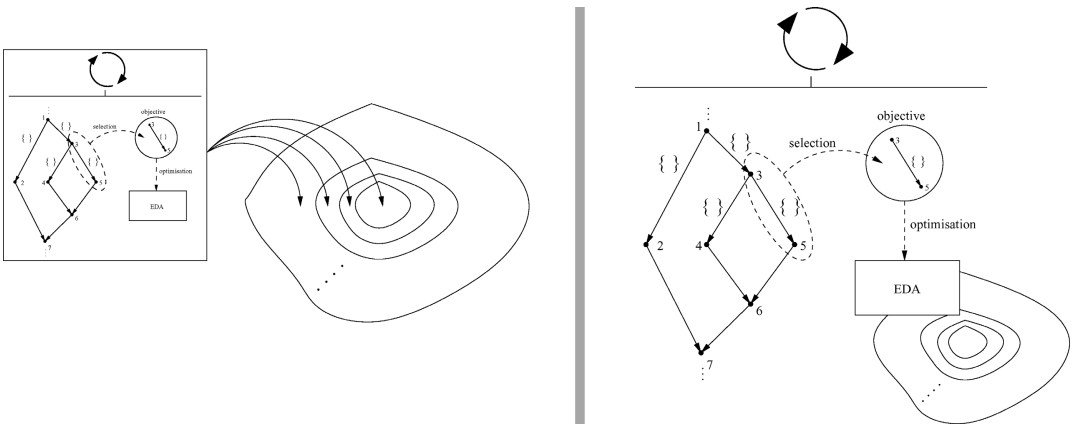

FIGURE 6. Schemas of the MOA (left-hand panel) and SOA (right-hand panel) approaches.

Obviously, this is a difficult task, because the topology of the space should be known in advance (and no search would be required then).

Instead, it is possible to approximate the problem by using static heuristic information from the program's source code. Although different source code aspects could be regarded, in the present work, this information is obtained from the expression in the conditional statement corresponding to a branch. Assuming, with no loss of generality, that an input is composed of three parameters (a, b, c), then, the center of the initial region may be elicited through the following two heuristic rules:

- If an expression follows the form $F(a, b, c)\,\mathbf{OP}\,K$, where F is a known function of the input parameters, K is a constant and **OP** is a comparison operator, then the region is centered at point (C_a, C_b, C_c) such that $F(C_a, C_b, C_c) = K$.
- If an expression follows the form $F(a, b, c)\,\mathbf{OP}\,F'(a, b, c)$, where F and F' are known functions of the input parameters, and **OP** is a comparison operator, then the region is centered at point (C_a, C_b, C_c) such that $F(C_a, C_b, C_c) = F'(C_a, C_b, C_c)$.

Notice that the above rules refer to specific types of expressions. Many possibilities exist for the form of functions F and F' in an expression. For instance, it could depend on a number of source code variables or it might include calls to other programs. These rules constitute a first approximation to the problem by restricting F and F' to depend only on the input parameters, e.g., $F(a, b, c) = 7a + 25c$. Furthermore, each point (C_a, C_b, C_c) was calculated manually for the experimental programs employed to evaluate the present work. To reach complete automation of this step, a numerical calculus tool could be employed, for example, Mathematica.[1]

In case none of the above rules can be applied, the center of the initial region for a branch is obtained through a heuristic strategy based on the program's dynamic information; to be exact, a grid search method is employed. For each input parameter, the complete range of values is partitioned into τ intervals. The center of each of these intervals is taken as a reference value. Then, the inputs resulting from the combination of the reference values of all the parameters are evaluated with regard to the branch. The best input is selected as the center of the initial region. Notice that the granularity of the strategy may be tuned with the number of intervals τ, because the number of inputs generated is τ^p for a program with p parameters. The idea behind a grid search is to explore a number of equally distant points from the whole search space. As τ grows, the number of points being considered approaches the complete number of points and, hence, the quality of the solution found might increase. On the other hand, reaching a certain value of τ may result in an unavoidable number of points. As a consequence, τ is regarded as a parameter of the approach. Figure 7 illustrates the strategy for the case of two parameters and $\tau = 6$; among the 6^2 points, the one inside the circle represents the input hypothetically chosen as the center.

Once the center is obtained using whichever of the strategies above, the specification of the initial search region of the branch is completed by defining an amplitude. This is achieved by setting, for each input parameter, an increment over the center. These initial increments are given as parameters to the test data generation system.

Thus, in essence, there are two types of branches. On one hand, those with the region centered at a point obtained through static heuristic information and, on the other hand, the branches with the region center chosen by means of a grid search, i.e., using dynamic information.

[1]Mathematica is a software package that solves equations symbolically. Web site: `http://www.wolfram.com/mathematica/`

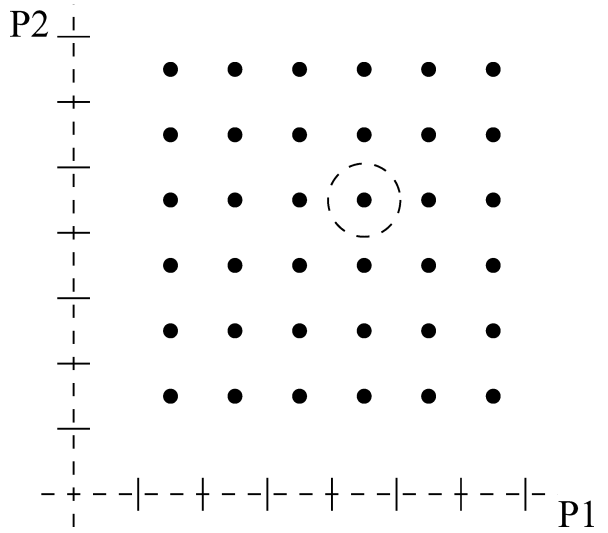

FIGURE 7. Schema of the grid search method.

4.3.1.2. Stopping Criteria—Steps 2 and 3 (MOA), Steps 2 and 4(SOA). The stopping criterion at step 3, for MOA, and step 2, for SOA, refers to the general scheme (Figure 2). It is defined in the same way as in the basic approach, that is, full coverage achievement or unsuccessful treatment of every uncovered branch.

In contrast, the criterion in step 2, for MOA, and step 4, for SOA, alludes to the Self-Adaptive approach. Therefore, it states the point where the search space stops being enlarged. To obtain this point, a limit to the size of the region is given as a parameter to the system. Accordingly, in the case of MOA, the stopping criterion is to obtain full branch coverage or reach the size limit, while in SOA, the search stops when the objective branch is covered or the space attains its size limit.

4.3.1.3. Branch Selection—Step 4 (MOA), Step 3 (SOA). The objective branch is selected following the strategy of the basic approach. Hence, the branch with the highest quality set of inputs at the moment is chosen, that is, the branch with the highest average objective function value over the inputs in the set.

4.3.1.4. EDA-Step 5 (MOA, SOA). The EDA seeks the optimal input in a search region centered at a fixed point. Therefore, an individual is a bit string representing an increment on the center of the current region. To be precise, the individual consists of a bit substring for each input parameter. Each substring represents an increment on the center of the interval of the corresponding parameter.

In the evaluation, the increment represented by the individual is added to the center of the region, resulting in the input for the objective function. In the current implementation of the approach, three parameter types are considered: integers, reals, and characters. In the case of an integer, the bit substring represents the increment following a sign-magnitude codification. For real numbers, the IEEE floating point codification is used instead. In both cases, the input parameter value is obtained by summing the increment to the center. Finally, for a character type, a sign-magnitude codification is employed again in the substring. Then, the increment is summed to the center of the parameter, and the value obtained results in a

character, according to the ASCII code table. Similarly, for more complex parameter types, an appropriate transformation could be defined to obtain the input parameter value.

As in the basic approach, the input is evaluated with regard to all the other uncovered branches and the sets of best inputs are updated accordingly.

The length of the individuals may vary between different EDA executions and, in consequence, it is not advisable to keep the same parameter values for the whole process. This is overcomed by making some of the parameters adaptive (Eiben et al. 1999).

A common practice in Evolutionary Algorithms is to fix the population size proportionally to the number of variables. For instance, in Mühlenbein and Mahnig (2001), several rules of thumb are suggested for a number of EDAs under specific conditions. In the present work, the population size is set at twice the length of the individual.

On the other hand, it would be desirable to halt the search when no improvement can be obtained. This is a relatively unexplored matter in the field of EDAs, although a few recent works are emerging (Ocenasek 2006). Here, a novel strategy has been developed. The problem is approximated by identifying the generation where the estimated probability distribution $p_l(\boldsymbol{x})$ is similar to the empirical distribution of the selected individuals. Thus, the criterion adopted is to stop the EDA when the Kullback–Leibler cross-entropy from $p_l(\boldsymbol{x})$ to $p(\boldsymbol{x})$ falls below a value α given as a parameter to the system.

4.3.1.5. Region Enlargement—Step 6 (MOA, SOA). The size of a search region is determined by the amplitude of the interval associated to each input parameter. In other words, this size is defined by a maximum increment on the center of the interval of each parameter. In the EDA, an increment for each parameter is represented as a substring of bits. Therefore, the number of bits in each substring specifies the size of the region.

The search region is enlarged by augmenting the amplitude of the interval associated with a chosen input parameter. A bit is added to the substring representing the next parameter in the order given by the input, from left to right.

4.3.2. Management of the Set of Inputs. The control of the set of inputs of each branch introduces disparities between the approaches which require a separate explanation.

4.3.2.1. Operation in MOA. In the MOA alternative, during the EDA execution, it is possible that an input being evaluated for a branch distinct from the objective falls outside the current search space. Therefore, when the branch is selected as the new objective and the EDA is to be initialized with the inputs in the set of the branch, some of these inputs might be out of the region.

Hence, instead of using only one set of inputs, two sets are associated with each branch. One of them keeps the best inputs inside the current search region—*inside set*—and the other one, those falling outside—*outside set*. This implies that, during the evaluation in the EDA, the input is stored in the required set and, this way, the initialization is directly performed from the inside set. More precisely, for each input in the set, the corresponding increment on the center is obtained (in its binary form) and added to the population.

To maintain the sets, before starting a new run of the general scheme (step 3), the inside set is updated with the inputs in the outside set which belong to the new region.

4.3.2.2. Operation in SOA. Regarding the SOA approach, each time the EDA executes the search region is different from the previous. In this situation no advantage is obtained with two sets, so just one containing all the inputs is used.

To initialize the EDA, firstly, the increments associated with the inputs in the set are calculated. Then, the increments inside the current region are included in the population. Those

falling outside are truncated to fit into the region and, then, are added to the population. A possible disadvantage of this strategy is that, as the population converges to similar individuals, if these are high quality solutions, they will be included in the set. Thus, initialization for the next region might cause a low diversity between individuals and result in a poor search. With the intention of alleviating this phenomenon, half of the EDA's initial population is randomly generated.

An other problem in SOA concerns the retrieval of the initial search region for the EDA. If the objective branch is selected for the first time, the initial region is given by its center and the initial increment. However, it can so happen that, in the EDA evaluation, the input enters the set of a branch already treated and, therefore, makes this branch a candidate objective once again. Supposing that the branch is selected for a second time, the initial search region should not be taken as before, because the new inputs in the set could be in a larger space and, hence, would not be used to seed the population. The solution adopted here has been to obtain the initial region size of the smallest new input in the set.

4.4. An Execution Example

As an illustration of the approach, some steps of an hypothetical execution of MOA and SOA are explained next. The example of Figure 3 will be used once again. Hence, test cases are to be generated for a program where an input is composed of three integers a, b, and c.

First of all, both algorithms require the assignment of an initial search region to each branch (step 1). Thus, for each branch and input parameter, an initial interval of values must be defined. This is attained by fixing the center of the interval and an increment on the center.

Two strategies are proposed for the center elicitation: static information and dynamic information based. The branch represented by arc (2, 3) in the graph is associated to condition `if((b*b) - (4*a*c) < 0)`, so the static atrategy is used. A point satisfying $b^2 - 4ac = 0$ is chosen as the center, for instance, (0, 0, 0). In contrast, the condition of branch (7, 8) is `if((a * x * x + b * x + c) == 0)`, so the grid search method must be employed. If an integer is codified with 15 bits in two's complement representation, the complete interval of values of each parameter is $[-32768, 32767]$. With $\tau = 8$, $8^3 = 512$ points are generated and evaluated. The best is (4095, 4095, −20480), which is taken as the center of the region.

Once the center of each branch is fixed for a, b and c, the initial region is obtained with an increment on each center. To keep the example simple, 5 bits are given to represent an increment for each input parameter, resulting in a maximum increment of ± 31. Thus, the initial region for branch (2, 3) is $[-31, 31] \times [-31, 31] \times [-31, 31]$ and for branch (7, 8) it is $[4064, 4126] \times [4064, 4126] \times [-20511, -20449]$.

4.4.1. MOA Example. MOA applies the basic approach (steps 3 to 5) over increasing search regions until a maximum size is achieved (step 2). Using a maximum of 10 bits to represent an increment for each input parameter, the maximum region for branch (2, 3) is $[-1023, 1023] \times [-1023, 1023] \times [-1023, 1023]$ and for branch (7, 8) it is $[3072, 5118] \times [3072, 5118] \times [-21503, -19457]$.

Now, assume that the size of the region in the current round is defined with 7 bits for a and b, and 6 bits for c. This implies that, in the previous region, 6 bits were used for b.

Remember that two sets of inputs are associated to each branch: the inside set and the outside set. The selection strategy (step 4) chooses the branch with the highest quality inside set. If branch (2, 3) was selected, the initial population of the EDA (step 5) is seeded with the inside set of this branch. In this particular case, an individual representing the increment (98, −34, 15) would result in input (98, −34, 15), as the region center is (0, 0, 0). Aside

from calculating the objective function value of this input, it is also evaluated for the rest of the branches. For instance, evaluating the input for branch (7, 8) implies that its associated increment must be induced. Thus, input (98, −34, 15) results in increment (−3997, −4129, 20495) for branch (7, 8). To represent such an increment, 12 bits would be necessary for parameters *a* and *b*, and 15 bits for *c*, so it falls outside the current region. In consequence, the outside set is updated if its quality is improved with this input.

Once the basic approach finishes without covering all the branches, the current region is enlarged. In the previous region the interval of *b* was increased, so now *c* is augmented to 7 bits, resulting in a search region where *a*, *b*, and *c* represent an increment with 7 bits.

4.4.2. SOA Example. In SOA, the optimization phase is applied over increasing regions (steps 4 to 6). The rest of steps are those in the basic approach, so they are not illustrated here. Nowon, the following is assumed. Branch (2, 3) is selected as the objective and the current region of the optimization phase is defined with 7 bits for *a* and *b*, and 6 bits for *c*.

In this algorithm, only one set of inputs is maintained for each branch during the process. Half of the EDA's initial population is randomly created and the other half is seeded from the inputs in the set. For instance, to seed the population with input (509, −11, 35), the increment associated to branch (2, 3) must be induced first. The result is increment (509, −11, 35). This increment falls outside the current region because 9 bits are needed to codify the 509. Therefore, the bit substring representing the 509 is truncated to 7 bits to fit in *a*'s interval. In contrast, an input (45, 117, −21) would result in the increment (45, 117, −21), which is inside the current region and is to enter directly in the initial population.

As in the basic approach, once the value of an input is obtained for branch (2, 3), it is evaluated for the remaining branches. If the quality of the set of the branch is improved, then the input enters the set.

After the EDA finishes, the current region is enlarged in the way described above for MOA.

5. PERFORMANCE EVALUATION

To observe the performance of the presented approaches, test cases were generated for a number of programs taken from the literature. The goal of the evaluation was threefold: analyzing the behavior of the approaches, comparing their results with those attained by other alternatives, and checking whether they constitute a solid alternative in the real world. Regarding the former goal, performance of each algorithm, MOA and SOA, was studied in isolation. In the second goal, three topics were considered. Firstly, MOA was compared to SOA. Then, the static information based heuristic employed to define the initial search region was compared to the dynamic one. Finally, MOA and SOA's results were faced to those by the basic test data generator. For the later goal, MOA and SOA were evaluated over a number of "real-world" programs.

5.1. Experimental Setting

The experiments comprised nine test programs which are commonly used for validation in the field. Although most of these programs implement relatively simple algorithms, their source codes include a number of challenging branches for a test data generator. Anyhow, difficultness of branch coverage depends on the source code, so the implementations used here were those used for experimentation in other works. Programs are outlined next.

`ClassifyTriangle` This is a popular program where 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 with the input. Five different versions were used. The `Triangle1` program (Wegener et al. 2001) owns three integer-valued parameters and 26 branches (objectives) to be covered. `Triangle2` (Wegener et al. 2001) is the same as `Triangle1` with floating point parameters instead. On the other hand, `Triangle3` (McGraw et al. 2001), `Triangle4` (Sthamer 1996) and `Triangle5` (Bueno and Jino 2002) are different implementations with integer-valued parameters and 20, 26, and 14 branches respectively.

`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; the number of branches is 30.

`Remainder` This function (Sthamer 1996) calculates the remainder of the division of two integers, therefore an input is composed of two integer-valued parameters; the source code reveals 18 branches.

`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. An input is formed by six integers and the number of branches is 22.

`Quotient` Given two integers, the `Quotient` program (Bueno and Jino 2002) calculates the quotient and the remainder of their division. An input consists of two integer-valued parameters and 10 branches exist in the source code.

In Sagarna and Lozano (2005a), several EDAs were evaluated for test data generation using the basic approach. After analyzing the results, it was concluded that the TREE algorithm showed the best performance overall. In consequence, TREE was the EDA chosen here for the optimization step in both MOA and SOA approaches. At each generation, half of the population was selected according to a rank-based strategy. New individuals were simulated by means of Probabilistic Logic Sampling (Henrion 1988), and the population was created in an elitist way. The objective function employed in the experiments was formulated according to equation (2). Notice that the stopping criterion adopted for the EDA seems to be specially suitable for TREE. This algorithm obtains the tree dependent factorization minimizing the Kullback–Leibler divergence to the empirical distribution. Because the EDA stops when this divergence value is lower than α, the value of the optimal model is directly being considered. For the experiments, α was determined after a number of preliminary executions.

Other system parameters that need to be fixed are the size of the initial and the largest possible region. Given a program, this is achieved by setting, for each input parameter, the minimum and maximum possible amplitude of its associated interval of values. Obviously, a different amplitude may be linked to each input parameter and, thus, the shape of search regions could be controled. However, for the experiments, no a priori knowledge is assumed and, therefore, amplitude values were kept constant for all the input parameters of a program.

Table 1 presents, for each program, the values selected for the system parameters, i.e., number of bits for the increment on the initial region (minimum), number of bits for the increment on the largest allowed region (maximum), and α value.

Also shown in Table 1 is the number of branches, and for how many of them the centers of each input parameter were obtained through the static information based (static) and the dynamic information based heuristic (dynamic). As it can be seen, in all the programs but three, most of the branches are static. `Remainder` and `Quotient` are relatively balanced in this sense, while in `Atof`, outstanding branches are dynamic.

Remember that the dynamic information based strategy consisted of a grid search. In this method, the value of parameter τ defines the number of inputs being considered candidate

TABLE 1. Experimental Programs Characteristics and Parameters in the Experiments

	Characteristics			Parameters		
Program	Branches	Static	Dynamic	Minimum	Maximum	α
`Triangle1`	26	24	2	5	15	2
`Triangle2`	26	24	2	5	7	2
`Triangle3`	20	16	4	5	10	2
`Triangle4`	26	20	6	5	10	2
`Triangle5`	14	10	4	5	10	5
`Atof`	30	2	28	5	7	25
`Remainder`	18	10	8	5	16	5
`Complexbranch`	22	18	4	5	10	15
`Quotient`	10	4	6	5	10	2

centers. More precisely, for a program with p parameters, τ^p inputs are created for evaluation. On the other hand, the larger the τ, the finer the granularity of the strategy and, hence, the chances of finding a high-quality initial search region increase. In the experiments, τ was set from 1 up to 5 for all the programs excepting `Atof`, which used τ up to 3.

Additionally, to avoid too long executions, a limit of 150,000 inputs generated was established. As soon as this limit was detected, the experiment was forced to terminate.

5.2. MOA Performance

Table 2 presents the results of the experiments conducted. For each value of τ and each program, the table collects the average values in ten executions for the percentage of covered branches (%) and the number of inputs generated during the process (#).

The coverage value is a main gauge of the performance of a test data generator. However, the number of inputs obtained reflects the effort made during the process. Therefore, it is important for a generator to obtain a coverage value with the lowest cost, that is, producing as few inputs as possible. This implies that, given two generators achieving the same coverage, the one yielding the fewest inputs is preferred. According to this, Table 2 highlights in gray the best values of τ for each of these two measures and each program.

As can be noticed, in all programs except `Atof`, full coverage is reached. `Atof` seems to be the hardest, because the lowest coverage and the largest number of inputs are attained in this program. By contrast, `Quotient` obtains 100% coverage with the smallest number of inputs, so it appears to be the easiest program.

5.2.1. Overall Performance Analysis. Regarding at Table 2, no apparent relation exists between τ and the best results, because these are obtained with alternative values of τ, ranging from the lowest to the largest value.

To validate the best performance values in MOA, an analysis based on statistical tests was conducted. Because coverage is a primary measurement, for each program and each value of τ, the Mann-Whitney test with regard to the best τ value (in gray) was applied to the coverage results. Then, for the cases where no difference was found, the test was again used over the number of inputs generated. Table 2 presents the outcomes of these tests; symbol "†" denotes the cases where coverage dissimilarities ($p < 0.01$) were found, while "‡" refers to the number of inputs.

TABLE 2. Results of the MOA Approach

	Triangle1		Triangle2		Triangle3		Triangle4		Triangle5		Atof		Remainder		Complexbranch		Quotient	
τ	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#
1	100	212	99.62	579	100	302	100	2223	100	1476	98.33	68936	99.44	629	95†	149952	100	25
2	100	282	99.62	995	100	338	100	1967	96.43	5062‡	43.33†	150062	98.89	1628	100	1856	100	121‡
3	100	190	99.23	1143	100	311	100	2436	96.43	8807‡	96.33	134849‡	100	186	95.45†	102711	100	131‡
4	100	440‡	99.23	880	100	285	100	1922	96.43	4306‡	-	-	100	84	100	16096‡	100	115‡
5	100	381‡	100	990	100	330	100	1738	97.14	6163‡	-	-	100	57	95.91†	117213	100	44‡

In less than half the cases, the best values of τ constitute an improvement with statistical evidence. It can be seen that statistically significant differences were obtained for the coverage reached in `Atof` and `Complexbranch` for a few values of τ. In contrast, in the number of inputs generated, dissimilarities were observed in `Triangle1`, `Triangle5`, `Atof`, `Complexbranch`, and `Quotient`. Hence, according to these results, it may be concluded that, in general, τ has no significant effect on the best coverage. This implies that the measure that might define the best τ for a program is the number of inputs. In fact, the bulk of the differences found correspond to this measure. However, the number of these dissimilarities is 12 from up to 30 possibilities, so it cannot be concluded whether the best τ makes a difference for the number of inputs.

5.2.2. Initial Region Heuristics Performance. According to the previous analysis, no clear conclusion can be stated on the most suitable τ value for a program. To better understand the relevance of τ in the results, it could be interesting to examine the influence of the initial heuristics used to elicit the initial regions.

Table 3 shows, for each program, the number of branches covered ($\#_o$) and number of inputs generated ($\#_i$) by the static and dynamic heuristics from the region initialization phase. The first row presents the values of the static heuristic, while the rest correspond to the dynamic heuristic (grid search) with the different values of τ. Notice that the overall contribution of these heuristics consists of the sum of the static and the dynamic results for a chosen τ. For instance, in `Triangle1` with $\tau = 2$, after applying the static and dynamic strategies, $2 + 1 = 3$ branches were covered (which implies a 11.54% coverage) and $26 + 8 = 32$ inputs were generated.

It can be seen that, regarding the static strategy, a number of branches are covered in all the programs just by the application of the two heuristic rules. Moreover, in some cases this is a significant number. In `Complexbranch`, 10 out of the 18 static branches are covered, and in `Atof`, one of the two static branches are attained. Anyhow, considering that most of the branches are static in the main body of the programs and that 100% coverage was obtained in almost all of them, the heuristic rules appear to be effective.

The dynamic heuristic is a grid search method. In such a method, given a problem, as τ increases, more points are generated and the quality of the best solution found is expected to grow. In the context of the test data generator, this implies that the number of branches covered is expected to increase with growing values of τ. However, a main drawback of a grid search is that the value of τ needed to reach an outstanding solution may be large, producing a prohibitive number of solutions. This could be the case even for small values of τ, if the number of problem variables is relatively big (Bäck 1996). Accordingly, Table 3 shows alternating behaviors. In `Triangle3`, `Triangle4`, `Remainder` and `Quotient`, the coverage increases as τ grows, while, for the rest of the programs, this is not held. Moreover, comparing the values in Table 2 and Table 3 for `Triangle1` and `Triangle3`, it can be observed that, with $\tau = 5$, a significant part of the inputs are generated by the grid method; the same occurs in `Atof` with $\tau = 3$. In consequence, results do not necessarily improve by increasing the value of τ.

This observation can also be extrapolated to the best overall results in Table 2, because these are obtained with different values of τ. Furthermore, recall that, in the previous statistical analysis, no significant influence of τ on the best coverage values was found, excepting a few cases. Thus, these results suggest that the effect of the grid search is neutralized by the rest of the phases in MOA.

5.2.3. Region Enlargment Performance. An other factor that contributes to the performance of the generator is the number of region enlargements. New regions may include

TABLE 3. Results of the Initial Region Obtainment Heuristics

	Triangle1		Triangle2		Triangle3		Triangle4		Triangle5		Atof		Remainder		Complexbranch		Quotient	
	$\#_o$	$\#_i$	$\#_o$	$\#_i$	$\#_o$	$\#_i$	$\#_o$	$\#_i$	$\#_o$	$\#_i$	$\#_o$	$\#_i$	$\#_o$	$\#_i$	$\#_o$	$\#_i$	$\#_o$	$\#_i$
static	2	26	2	26	1	20	1	26	2	14	1	30	1	18	10	22	1	10
dynamic, $\tau = 1$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
dynamic, $\tau = 2$	1	8	0	8	2	8	3	8	0	8	0	1024	7	4	2	64	5	4
dynamic, $\tau = 3$	0	27	0	27	2	27	3	27	0	27	0	59049	7	9	2	729	5	9
dynamic, $\tau = 4$	1	64	0	64	3	64	6	59	0	64	-	-	8	14	2	4096	5	16
dynamic, $\tau = 5$	1	125	0	125	3	125	6	119	0	125	-	-	8	22	2	15625	6	24

uncovered objectives. Instead, as more increments are carried out, the number of inputs created is expected to grow, because more search steps are executed.

During the experiments, each run was recorded with the purpose of studying how the system operates. Using this information, given a search region and an objective branch, the number of inputs generated and whether the objective was covered or not can be elicited. This is shown in Figure 8. The graphics above relate to the number of branches covered in each region. More specifically, they only consider the branches which were covered by the initial region heuristics or those selected as objectives and covered by the EDA. Notice that not all the branches need to be explicitly searched, because during the fitness evaluation in the EDA, branches distinct from the objective may eventually be covered. Thus, in each graphic ahead of Figure 8, the x-axis takes values in the range of possible regions, while the y-axis concerns the number of branches covered by the initial heuristics or by the EDA. The points depicted correspond to the results (averaged over the ten executions) in each region, for each value of τ. To finish with the specification, Table 4 shows the average total number of branches searched by the EDA; aside from a program name, the number of branches in the program is provided in brackets. Analogously, the bottom part of Figure 8 presents the accumulated average number of inputs generated (y-axis) for each region (x-axis), given a value of τ and a program.

As can be observed in the upper half, with the exception of `Atof`, almost all the branches were attained in the very first search regions. To some extent, this is not surprising, because the first region includes the coverage of the initial heuristics and the EDA, while the rest of regions only involve the EDA contribution. In the bulk of the programs, the number of static objectives is high (see Table 1), so the graphics suggest that the static information based heuristic used to elicit the initial region is an adequate strategy. Indeed, this could be the cause of the poor behavior of `Atof`, because it contains a reduced number of static branches. Moreover, owing to the quite large set of parameters of an input in this program, τ only takes values up to 3, which seems to be insufficient for the grid search to obtain a promising initial center. Other programs with a relevant number of dynamic branches are `Remainder` and `Quotient`. In these cases, both the dynamic and the static heuristics appear to behave successfully, as all the dynamic branches were covered directly by the grid method (see also comments on Table 3) and most of the static ones were attained in the initial region. Anyhow, the effect of the different search spaces should not be underestimated. It can be noticed that, in 7 of the 9 programs, a few objectives are still covered in advanced regions and, therefore, the coverage measurement grows.

As for the inputs generated, Figure 8 below shows that their number stays relatively low at the initial stages, although it increases as branches remain uncovered. If complete coverage is attained, the curve stabilizes, in other case, it keeps growing. More especifically, the curve grows smoothly in a number of cases (e.g., `Triangle2`), although for other instances it augments rapidly with certain values of τ (`Complexbranch` and $\tau = 1$, for example). In these last cases, the latter regions offer more promising solutions than in the previous stages and the search intensifies. This means that the EDA operates for a larger number of generations and, thus, more solutions are generated. This can be clearly remarked in the `Triangle5`, `Remainder` and `Complexbranch` programs. The low coverage reached by `Atof` for $\tau = 2$ can be understood by observing the number of inputs generated. The figure reveals that the limit of 150,000 inputs was attained in the early regions, so the generator stopped prematurely and no more objectives could be covered (see `Atof` above).

To summarize, it could be deduced that, on one side, the search over different regions allows the MOA generator to obtain the highest coverage (effectiveness). On the other side, the answer of the dynamic heuristic seems to be more unstable than for the static information based strategy. In fact, the high-quality values of the early spaces suggest that the static

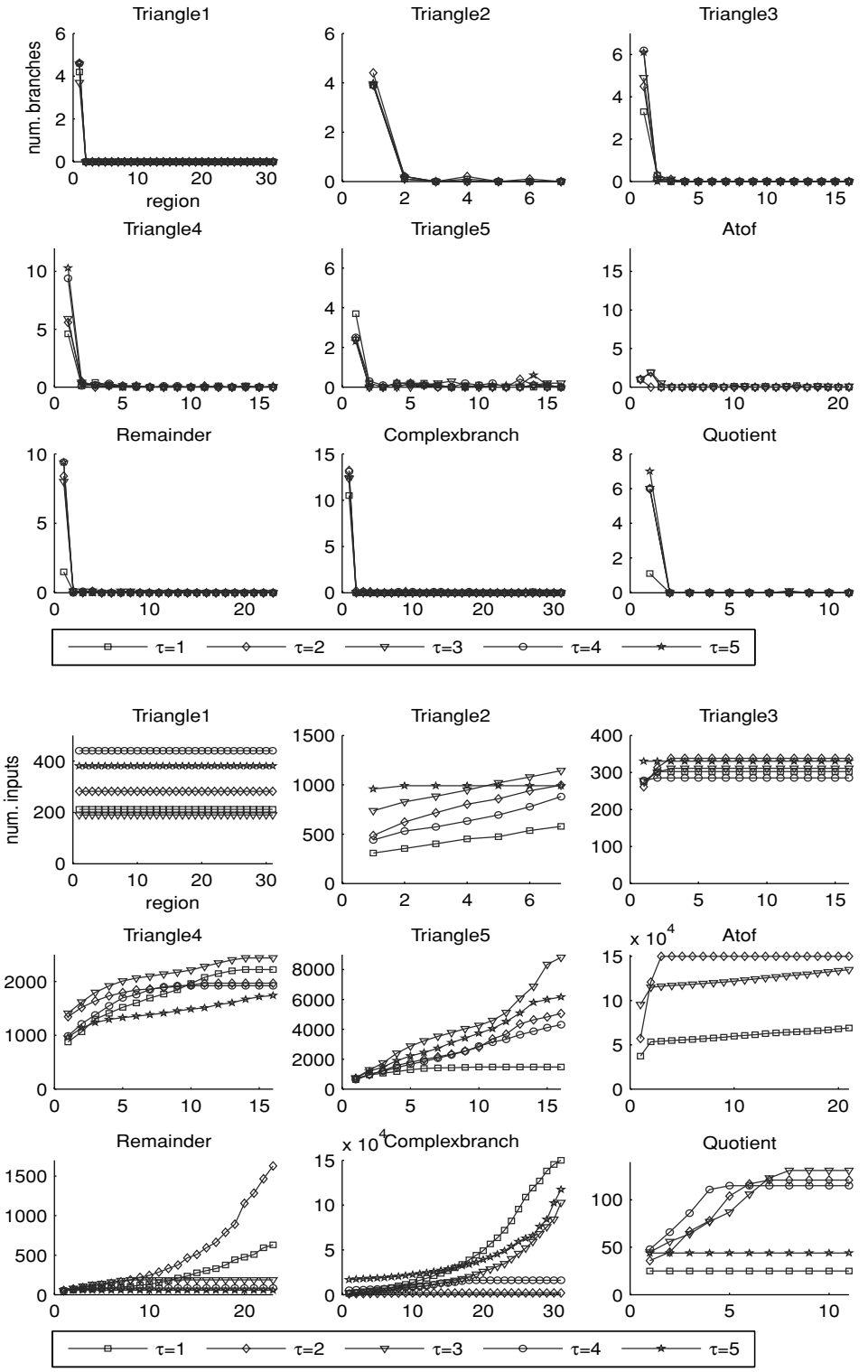

FIGURE 8. Average number of branches covered (above) and inputs generated (below) for each region in MOA.

TABLE 4. Average Number of Branches Sought in the MOA Approach

τ	Triangle1 (26)	Triangle2 (26)	Triangle3 (20)	Triangle4 (26)	Triangle5 (14)	Atof (30)	Remainder (18)	Complexbranch (22)	Quotient (10)
1	2.2	2.3	2.7	5.1	4.6	15.2	0.7	2.5	0.1
2	2	3.4	1.9	5.3	3.5	17	0.8	1.9	0.1
3	1.7	3	2.1	4.9	3.6	15.5	0.5	1.5	0.1
4	2.4	2.5	2.4	4	3.2	-	0.6	2.1	0.2
5	1.9	2.8	2.1	4.7	2.9	-	0.4	1.6	0

heuristic is useful to achieve objectives soon and, therefore, generate a reduced number of inputs (increase efficiency). To shed more light on this matter, this will be further studied in a following analysis in Section 5.4.

5.3. SOA Performance

Apropos the SOA algorithm, Table 5 shows the results of the experiments for the programs. The cell format is the same as in Table 2. Similarly to the MOA approach, the most difficult program for the test case generator is `Atof`. However, in this case, 100% coverage could not be obtained for `Triangle4` or `Triangle5`, either. Once again, the easiest program seems to be `Quotient`.

5.3.1. Overall Performance Analysis. Statistical tests were used to identify the best performance values. Thus, the null hypothesis of equal distribution densities between the best τ values and the others was evaluated in the manner explained in the previous section.

Differences were statistically significant ($p < 0.01$) with regard to the coverage measurement in a pair of cases (`Atof` with $\tau = 2$ and `Complexbranch` with $\tau = 1$). In contrast, for the number of inputs generated, the 18 differences obtained ($p < 0.01$), from up to 32 possibilities, spread over all the programs. The outcomes from this analysis reinforce the conclusions of the MOA approach. Taking the programs used here into account and respecting the best results, the τ value has no significant influence regarding the coverage measure. On the other hand, for the number inputs, not enough dissimilarities to make a reliable conclusion were found.

5.3.2. Initial Region Heuristics Performance. MOA and SOA share the same initial region elicitation step. Therefore, results in Table 3 also apply here, as well as the comments on the behavior of the static and dynamic heuristics.

Concerning the lack of influence of τ on the best coverage results, the outcomes of SOA are almost equal to those in MOA. In consequence, here, the corresponding reason is suggested, that is, the remaining steps of SOA cancel the effect of the grid search.

5.3.3. Region Enlargment Performance. Accordingly to the MOA alternative, the experiment executions were monitored and the values raised by a search step were stored. Figure 9 reveals, for each possible search region, the average number of objectives covered by the initial heuristics and the EDA (above), and the inputs generated (below) during the process. The figure format is the same as in the previous section. Table 6 assists in the understanding of the figure by presenting the average total number of branches searched by the EDA and, in brackets, the number of branches in a program.

Drawing a rough comparison of Figure 9 and Figure 8, it can be noticed that, in general, the behavior of both approaches is similar. Although differences appear with some programs (`Remainder` in the number of inputs), the remarks on the MOA algorithm can also be applied to SOA.

5.4. MOA vs. SOA vs. Other Approaches

Next, each Self-Adaptive algorithm is compared to other approaches to evaluate its performance and know if it represents a competitive alternative.

TABLE 5. Results of the SOA Approach

	Triangle1		Triangle2		Triangle3		Triangle4		Triangle5		Atof		Remainder		Complexbranch		Quotient	
τ	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#
1	100	401	100	333	100	250	99.23	3630	97.86	1697	96	65078	100	58‡	95.91†	122813	100	21
2	100	282	100	374	100	237	98.46	5476	97.86	4746‡	49.33†	150049	100	33	100	18406	100	24‡
3	100	246	100	293	100	222	99.23	6003‡	95.71	5886‡	95	133649‡	100	30	98.18	107144‡	100	123‡
4	100	391	100	266	100	297	98.46	2663	97.14	4334‡	-	-	100	59‡	98.18	68485	100	79‡
5	100	399‡	100	765‡	100	314‡	99.23	2546	97.14	3850‡	-	-	100	91‡	97.73	106698‡	100	44‡

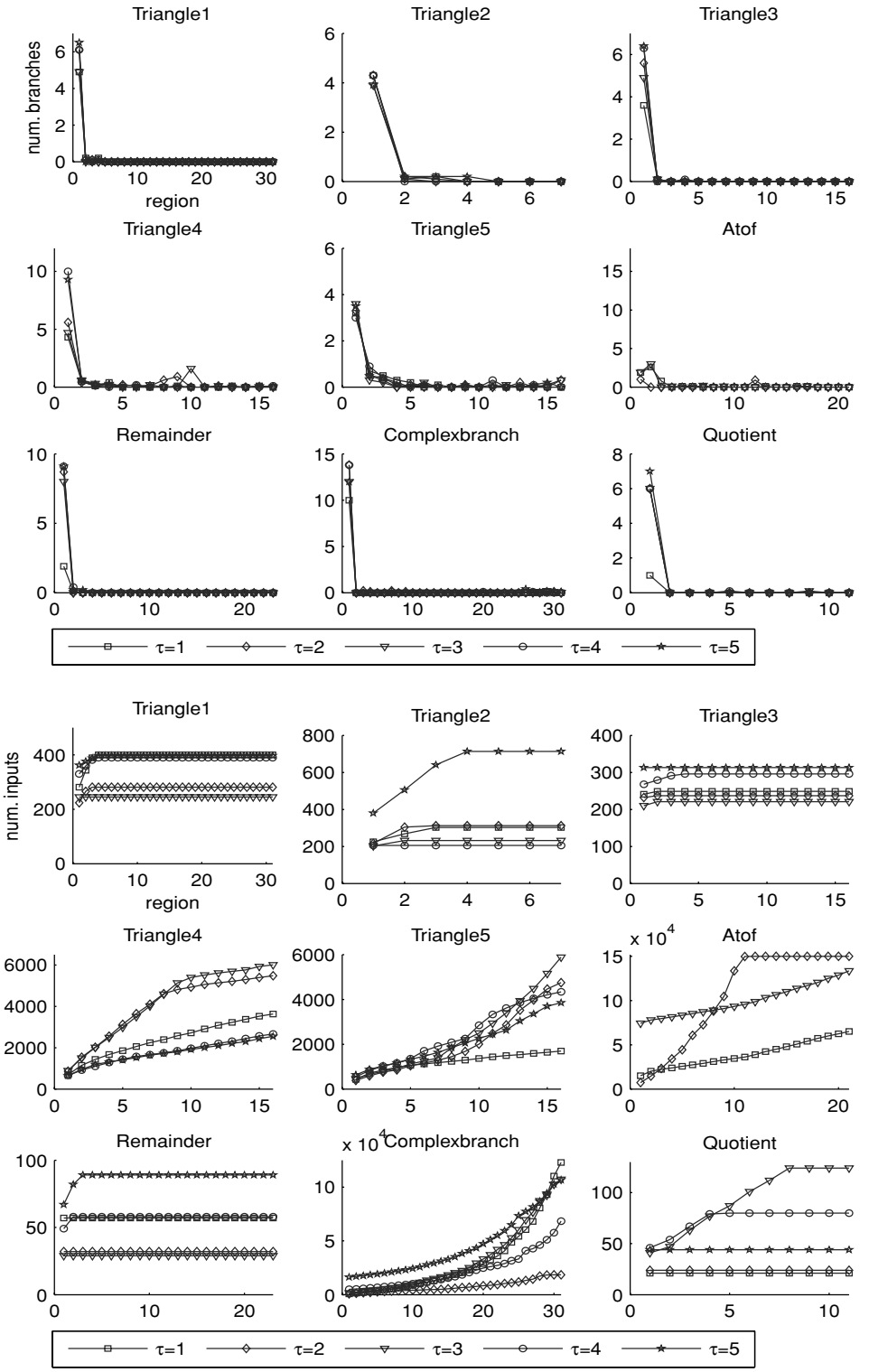

FIGURE 9. Average number of branches covered (above) and inputs generated (below) for each region in SOA.

TABLE 6. Average Number of Branches Sought in the SOA Approach

τ	Triangle1(26)	Triangle2(26)	Triangle3(20)	Triangle4(26)	Triangle5(14)	Atof(30)	Remainder(18)	Complexbranch(22)	Quotient(10)
1	3.4	2.6	2.7	5.1	3.4	6.7	1	1	0
2	3.3	2.6	2.7	4.9	3.2	1.9	0.7	2.6	0
3	2.9	2	2	4	3.3	6.6	0	1	0.1
4	3.3	2.3	2.4	4.4	3.5	-	0.5	2.5	0.1
5	3.6	2.5	2.4	3.8	3.6	-	0.5	1	0

5.4.1. MOA vs. SOA. In the MOA approach, each region enlargement concentrates on the test case generator as a whole. In contrast, each increment of the SOA alternative refers to an independent EDA search phase. Therefore, a formal comparison of both algorithms in terms distinct from the coverage and inputs generated becomes a difficult task. However, it might be suspected from the common conclusions raised in Sections 5.2 and 5.3, and from the matching behavior shown in Figures 8 and 9, that important similarities exist between them.

To know whether MOA and SOA offer a similar behavior in terms of coverage and inputs created, Table 2 and Table 5 were used to find statistically significant differences between the results. To be precise, the Mann-Whitney non-parametric test was applied to each approach and value of τ. Considering coverage, the null hypothesis of equal distributions was rejected ($p < 0.01$) only for `Atof` with $\tau = 2$ and `Complexbranch` with $\tau = 3$. For the number of inputs generated, differences ($p < 0.01$) were obtained in six cases: `Triangle1` with $\tau = 1$, `Triangle2` with $\tau = 3$, `Triangle3` with $\tau = 3$, `Triangle4` with $\tau = 2$ and $\tau = 3$, and `Remainder` with $\tau = 3$. Because half of the best result values in these cases corresponded to each approach, it cannot be stated which one behaves better.

According to the tests, it may be concluded that, excepting a few cases, the performance of MOA and SOA algorithms is similar in terms of coverage and number of inputs generated.

5.4.2. Static vs. Dynamic Information Centers. An element which appears to be important in the Self-Adaptive approach is the initial search space. If this is located in an adequate region, the effort in finding the optimal solution may be low. On the other hand, if the EDA departs from an unsuitable region, a huge number of interval increments could be necessary to reach the optimum, or it could not even be attained. In the present work, the definition of the initial space of each branch is based on static or dynamic heuristic information. To compare these two strategies, the previous experiments were repeated changing static information based centers to be dynamic information based. Tables 7 and 8 show the results for the MOA and SOA algorithms, respectively.

The differences between the static and dynamic strategies for the coverage and number of inputs were studied through statistical tests. In other words, the Mann-Whitney test was employed to evaluate the equality between the distribution densities of the algorithms with and without static strategy. Similarly to previous tables, the symbols "†" and "‡" beside a cell in Table 7 denote a statistically significant difference ($p < 0.01$) between the experiments in the cell and the corresponding values in Table 2. Analogously, the same applies to Tables 8 and 5.

As can be observed, in MOA, the differences associated with the coverage concentrate on three programs: `Triangle1`, `Triangle2`, and `Triangle4`. However, concerning the number of inputs generated, from up to 43 tests, dissimilarities were obtained in 27 cases. All in all, the programs with a large proportion of static to dynamic branches (see Table 1) offered differences, excepting `Complexbranch` for a few values of τ which shown an inferior performance in Table 2. In contrast, the programs with a more significant number of dynamic branches, revealed, in general, fewer dissimilarities.

In the SOA algorithm, coverage differences were found not only in `Triangle1`, `Triangle2`, and `Triangle4`, but also in `Triangle5` with $\tau = 2$. Similarly, for the number of inputs created, the same differences as in MOA were observed, removing `Complexbranch` with $\tau = 4$, and adding `Triangle3` with $\tau = 1$.

Regarding these significantly different instances in Tables 2–8, it can be noticed that in almost all of them, the best results correspond to the approach using the static strategy. The only exceptions are `Triangle3` with $\tau = 4$ and $\tau = 5$, and `Remainder` with $\tau = 5$ for both MOA and SOA, in the number of inputs generated.

TABLE 7. Results of the MOA Approach with No Static Information Based Initial Centers

	Triangle1		Triangle2		Triangle3		Triangle4		Triangle5		Atof		Remainder		Complexbranch		Quotient	
τ	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#
1	96.15†	150032‡	100	1688‡	100	453	100	1497	97.86	2368	99	89230	99.44	838	95.45	145272	100	71‡
2	76.92†	150024‡	11.54†	150024‡	100	5324‡	94.62†	64131‡	92.86	4949	43.33	150050	99.44	7825‡	90.91†	150047‡	100	2576‡
3	11.54†	150020‡	11.54†	150017‡	99	7829‡	88.85†	44311‡	92.86	5950	92.67	138653	100	49	95.45	112269	100	1232‡
4	100	61916‡	11.54†	150018‡	100	154‡	93.85†	29048‡	93.57	3723	-	-	100	6726‡	97.73	114918‡	100	1806‡
5	100	74577‡	11.54†	150024‡	100	185‡	91.54†	37602‡	95.71	5340	-	-	100	42‡	95.45	134630	100	44

TABLE 8. Results of the SOA Approach with No Static Information Based Initial Centers

	Triangle1		Triangle2		Triangle3		Triangle4		Triangle5		Atof		Remainder		Complexbranch		Quotient	
τ	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#
1	94.62†	150026‡	100	2314‡	100	376‡	98.46	3778	97.86	2107	95	91148	100	71	98.18	87891	100	51‡
2	76.92†	150023‡	23.08†	150024‡	100	3691‡	89.62†	28819‡	92.86†	3855	47.33	150064	100	9931‡	93.64†	128868‡	100	1304‡
3	56.15†	150028‡	34.62†	150020‡	99	4442‡	89.23†	28566‡	93.57	3626	95.67	125487	100	33	97.27	111895	100	956‡
4	100	25031‡	65†	145067‡	100	125‡	89.23†	24322‡	93.57	3645	-	-	100	8377‡	96.36	113230	100	938‡
5	100	30808‡	23.08†	150014‡	100	185‡	90†	25134‡	95	3676	-	-	100	42‡	98.18	116060	100	44

The remarks from these tests are captured by Figures 10 and 11 for MOA and SOA, respectively. In each graphic, the different objectives are represented in the x-axis, while the y-axis takes values in the range of possible region enlargements. Thus, given a program, the graphic in the upper half in a figure shows the number of increments performed for each static (labeled with a cross) and dynamic (labeled with a circle) objective. To be exact, the average and the standard deviation over τ and the ten executions are depicted for each objective. Analogously, in the bottom part of a figure, the values associated with the variant using only dynamic objectives are presented.

Both figures show clear disparities between the *static-dynamic* and the *dynamic* approaches in programs where the bulk of the statistical tests observed differences (`Triangle1`, `Triangle2`, `Triangle4`). In contrast, in programs with no significant dissimilarity (`Triangle5`, `Atof`), behavior is almost the same. Remaining programs fall somewhere in between; they respond differently for a few objectives, although, in most of them, response is alike.

The significant differences obtained in the number of inputs generated are also reflected by the figures. In all the statistically distinct programs, the sum of the average number of increments in the dynamic approach is larger than in the static-dynamic one. Indeed, it can be noticed that the main body of the objectives where changes occur between both approaches corresponds to static cases which had turned out to be dynamic.

Thus, it may be concluded that the suggestions raised in Section 5.2 on the static information based strategy are confirmed. This strategy can make a difference in the coverage reached but, most of all, in the number of region enlargements and, consequently, in the number of inputs created. Moreover, the static heuristic improves or equals the dynamic one, with the exception of a few cases.

5.4.3. Self-Adaptive vs. Basic Approach. To have an idea of the quality of the results of the Self-Adaptive alternative, they were compared with those obtained by the basic test data generator.

The range of input parameter values for the basic approach was obtained centering the interval in 0 and adding the maximum increment shown in Table 1. To make the comparison as fair as possible, the EDA chosen was TREE and its parameters were the same as in Section 5.1, apart from two of them. For the `Triangle5` and `Quotient` programs, the population size and the maximum number of generations were determined after preliminary experimentation; the values selected were 600 individuals and 200 generations in the former, and 200 individuals and 50 generations in the latter. For the rest of the programs, these parameters were fixed with the values in Sagarna and Lozano (2005a) offering the best performance for TREE.

Table 9 shows the best values (with priority to coverage) of the MOA, SOA and basic approaches. The outstanding results are highlighted in gray.

It can be observed that the Self-Adaptive alternative outperforms the basic approach in the coverage reached as well as the number of generated inputs in all the programs except `Atof`. In fact, the poor behavior shown in the results of previous tables for this program becomes evident here, mostly with regard to the number of inputs. In `Atof`, a number of objectives can only be covered when the largest search region is reached. Because the Self-Adaptive approach departs from a reduced region and the grid search method seems to provide an unsuitable initial center, performance is worse than for the basic alternative, which operates over the largest region directly.

The purpose of the current comparison is to identify the approach offering the best performance. Hence, the statistical analysis explained in Section 5.2 was used to validate these results. Similarly to the previous table, Table 9 provides the outcomes of the analysis.

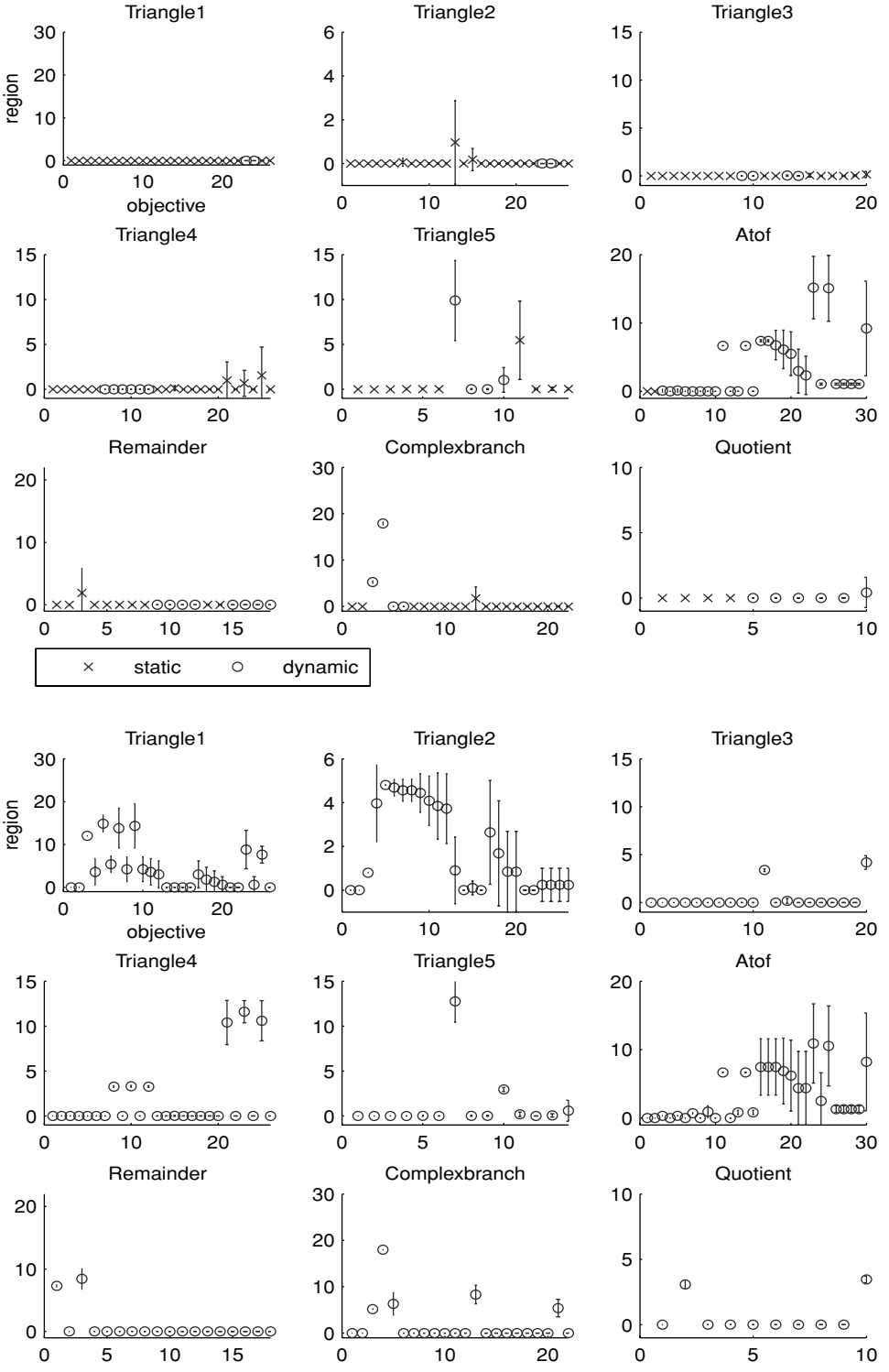

FIGURE 10. Average number of region enlargements per objective in MOA (above) and MOA with no static objective (below).

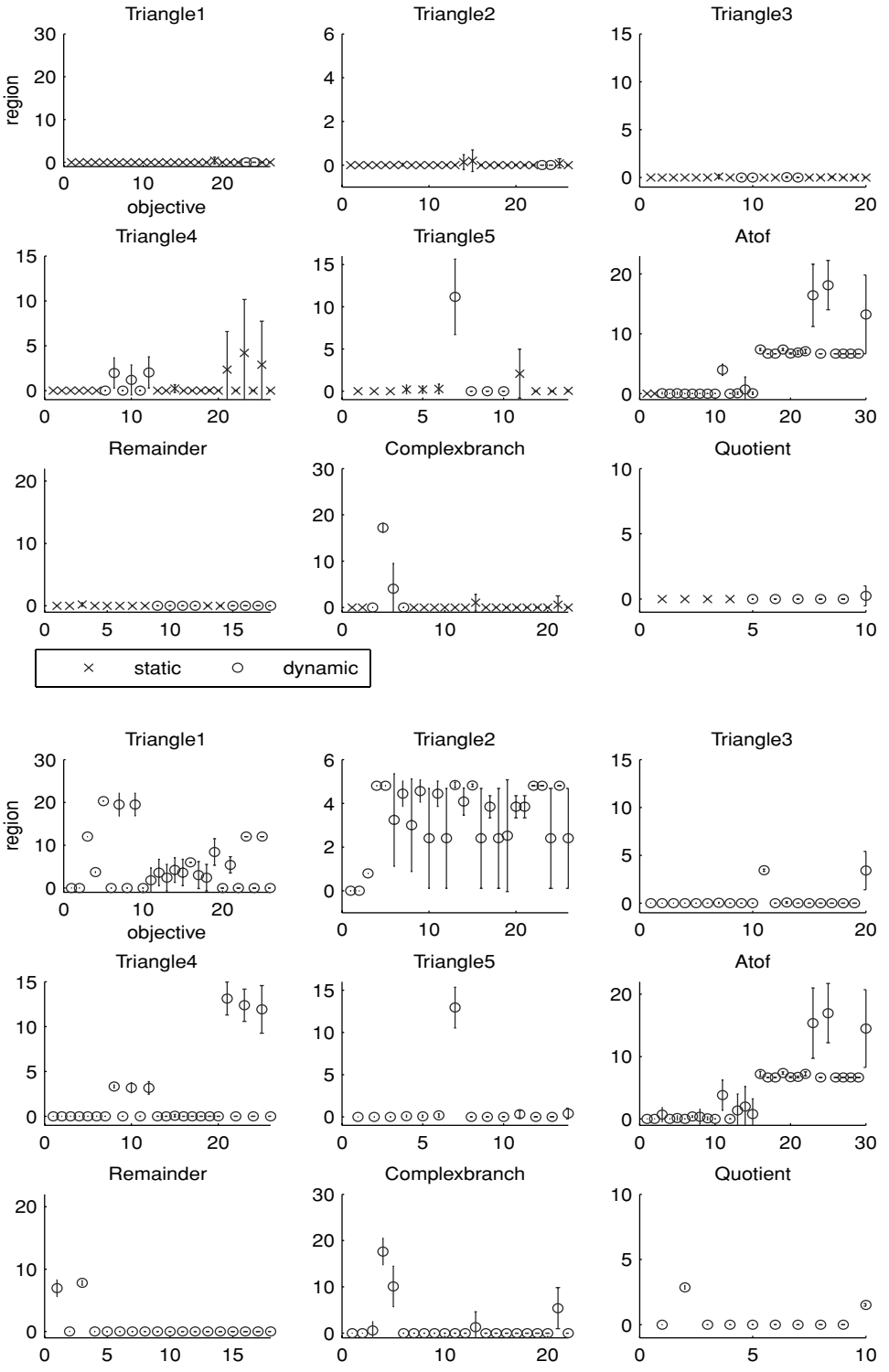

FIGURE 11. Average number of region enlargements per objective in SOA (above) and SOA with no static objective (below).

Significant differences ($p < 0.01$) in the coverage values were noticed just for `Atof`, between the basic approach and SOA. In contrast, MOA revealed a difference in this program for the number of inputs generated. Thus, it can be deduced that the basic generator improves SOA and MOA with statistical evidence in `Atof`. In spite of this, for the rest of the programs, the basic approach presents dissimilarities ($p < 0.01$) in the number of inputs created with regard to the best result.

Therefore, it can be inferred that in almost all the programs the Self-Adaptive approach outperforms the basic one.

5.5. Evaluation with Real-World Programs

The experiments conducted in the previous sections involve typical programs which are known to include several challenging branches. Obviously, test data generation for "real-world" programs may be as difficult, although it could result in a simple task as well. To verify whether the Self-Adaptive algorithms constitute a solid option in the "real world," they were compared to the basic approach for a number of non-academic programs. In Sagarna and Lozano (2008), test cases were generated with the basic approach for several programs taken from the book "Numerical Recipes in C. The Art of Scientific Computing." (Press et al. 1988). Thus, the Self-Adaptive alternative was applied on 16 instances randomly chosen from this study.

Apropos the parameters for the basic approach in Sagarna and Lozano (2008), the EDA applied was TREE. The population consisted of 100 individuals, and the stopping criterion was reaching a maximum of 100 generations. The rest of the parameters in the EDA were the same as in Section 5.1. Additionaly, the test case generation was halted as soon as a limit of 100,000 inputs was detected.

In the experiments with the Self-Adaptive approach, the EDA took the parameter values previously described, with two exceptions. As explained in Section 4.3.1, the EDA's population size is fixed to be twice the length of the individual. Moreover, to make a fair comparison, instead of using the Kullback–Leibler divergence based stopping criterion, a maximum number of generations equal to the population size was set. Again, the whole process was forced to terminate when the generation of 10,0000 inputs was detected. In all the programs, the parameters of an input were integers or real numbers. Tentative values were adopted for the number of bits used to represent the initial and the final search regions, i.e., 5 and 10 bits for integers, and 5 and 7 bits for real parameters.

The experiments were conducted for MOA and SOA, with τ ranging from 1 to 5. Table 10 presents the results of the best τ for each algorithm, together with the values of the basic approach. The outstanding values are highlighted in gray.

In all the programs but one, MOA or SOA improve the outcomes of the basic approach. In this exception (`cyfyn`), the basic generator obtained a 75% coverage and stopped at 40,100 inputs. The Self-Adaptive algorithms were unable to attain a better coverage, but they continued the search over larger regions until the maximum limit of inputs was reached. Athough this behavior results undesirable in this case, it can also be very suitable. For instance, in `bnldev`, the coverage of the basic approach is augmented and the limit of 100,000 is attained once again. The other programs where the coverage is outperformed are `caldat`, `flmoon`, and `tridag`. For the rest of the cases, the enhacement corresponds to the number of inputs generated.

Thus, these outcomes present the Self-Adaptive approach as a viable alternative for application in the real world. Furthermore, the results clearly support the conclusion from the previous section: the Self-Adaptive algorithms perform better than the basic approach, mainly with regard to the number of inputs generated.

TABLE 9. Best Results of the Basic, MOA, and SOA Approaches

	Triangle1		Triangle2		Triangle3		Triangle4		Triangle5		Atof		Remainder		Complexbranch		Quotient	
Approach	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#
Basic	99.62	9940‡	100	5880‡	100	4180‡	99.62	51560‡	99.29	47160‡	100	3475	100	1240‡	98.64	21420‡	100	520‡
MOA	100	190	100	990‡	100	285	100	1738	100	1476	98.33	68936‡	100	57‡	100	1856	100	25
SOA	100	246	100	266	100	222	99.23	2546	97.86	1697	96†	65078	100	30	100	18406	100	21

TABLE 10. Results of the Basic Approach, MOA, and SOA on Real World Programs

	Basic		MOA		SOA			Basic		MOA		SOA	
Program	%	#	%	#	%	#	Program	%	#	%	#	%	#
bessj	100	220	100	21	100	45	irbit2	50	10100	50	1511	50	1476
bnldev	80.77	54100	84.62	100007	84.72	100018	kendl1	100	100	100	61	100	61
caldat	75	20100	87.5	1550	87.5	1481	laguer	100	3590	100	2149	100	2185
cyfun	75	40100	75	100009	75	100011	ran1	66.67	20100	66.67	4730	66.67	3649
factln	87.5	10330	87.5	1543	87.5	1477	ratint	100	330	100	163	100	74
fit	100	3760	100	101	100	101	sncndn	93.75	10100	93.75	3089	93.75	3021
flmoon	98.33	2530	100	29	100	29	tred2	100	240	100	61	100	61
gasdev	75	10100	75	1541	75	1476	tridag	91.25	10790	100	157	100	157

6. CONCLUSIONS

In this paper, two new Search Based Software Test Data Generation approaches for branch coverage were described, namely, MOA and SOA. To enhance the test case generation process, the optimization step of both alternatives departs from an initial small region which is increasingly enlarged as branches remain uncovered. The starting search space is defined upon heuristic information from the program. More precisely, two options could be adopted: the application of a set of rules concerning the source code's static information, or using a heuristic procedure based on dynamic information, which consisted of a grid search method.

The analysis of the experiments conducted revealed promising results for both approaches. First of all, the search over different regions allows for the achievement of the highest coverage values, which is a primary performance measurement in test data generation. Apropos the two heuristic strategies to obtain the initial region, it was concluded that the static option makes a difference and can at least improve the efficiency of the approach in terms of the number of inputs generated. On the other hand, the dynamic heuristic showed to be more unstable. The τ parameter of the method did not provide a relevant influence on the best coverage values, in general. Instead, it can make a difference for the number of inputs, although no definitive conclusion could be stated on this.

Comparing the performance of the MOA and SOA algorithms, in general terms, no significant difference was found between them. Additionally, the algorithms were compared to the basic approach. With the exception of the poor results in one test program, the formers outperformed the latter with statistical evidence. Moreover, this improvement over the basic generator repeated for a number of "real-world" programs, presenting the Self-Adaptive strategy as a solid alternative.

By the way, several elements of the approach need to be further studied. For instance, in the dissapointing experimental results, almost all the initial regions where created in a dynamic way. Because the static strategy behaves superiorly, a way to enhance the response of the Self-Adaptive approach could be to expand the set of heuristic rules. To make the approach more flexible, another interesting line of future work is the elicitation of an α value for the stopping criterion of the EDA, which takes into account the size of the search space.

ACKNOWLEDGMENTS

This work was supported by the ETORTEK and SAIOTEK projects of the Basque Government, as well as the TIN2005-03824 project of the Spanish Ministry of Education and Science. R. Sagarna received additional support from the Department of Education, Universities and Research of the Basque Government under the program of Researcher Education.

REFERENCES

BÄCK, T. 1996. Evolutionary Algorithms in Theory and Practice. Oxford University Press, New York.

BALUJA, S., and S. DAVIES. 1997. Combining Multiple Optimization with Optimal Dependency Trees. Technical Report CMU-CS-97-157. Carnegie Mellon University, Pittsburgh, PA.

BARESEL, A., and H. STHAMER. 2003. Evolutionary testing of flag conditions. *In* Proceedings of the Genetic and Evolutionary Computation Conference, *Edited by* E. Cantú-Paz, J. A. Foster, K. Deb, L. Davis, R. Roy, U. O'Reilly, H. Beyer, R. K. Standish, G. Kendall, S. W. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, N. Jonoska, and J. F. Miller. Springer-Verlag, Berlin, pp. 2442–2454.

BEIZER, B. 1990. Software Testing Techniques. Van Nostrand Rheinhold, New York.

BOTTACI, L. 2003. Predicate expression cost functions to guide evolutionary search for test data. *In* Proceedings of the Genetic and Evolutionary Computation Conference, *Edited by* E. Cantú-Paz, J. A. Foster, K. Deb, L. Davis, R. Roy, U. O'Reilly, H. Beyer, R. K. Standish, G. Kendall, S. W. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, N. Jonoska, and J. F. Miller. Springer-Verlag, Berlin, pp. 2455–2464.

BUENO, P. M. S., and M. JINO. 2002. Automatic test data generation for program paths using genetic algorithms. International Journal of Software Engineering and Knowledge Engineering, **12**(6):691–709.

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, New York.

CHOW, C., and C. LIU. 1968. Approximating discrete probability distributions with dependence trees. IEEE Transactions on Information Theory, **14**(3):462–467.

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.

DÍAZ, E., J. TUYA, and R. BLANCO. 2003. Automated software testing using a metaheuristic technique based on tabu search. *In* Proceedings of the 18th IEEE International Conference on Automated Software Engineering. IEEE CS Press, pp. 310–313.

DURAN, J. W., and S. C. NTAFOS. 1984. An evaluation of random testing. IEEE Transactions on Software Engineering, **10**(4):438–444.

EIBEN, A. E., R. HINTERDING, and Z. MICHALEWICZ. 1999. Parameter control in evolutionary algorithms. IEEE Transactions on Evolutionary Computation, **3**:124–139.

FENTON, N. E. 1985. The structural complexity of flowgraphs. *In* Graph Theory with Applications to Algorithms and Computer Science, *Edited by* Y. Alavi, G. Chartrand, L. Lesniak, D. R. Lick, and C. E. Wall. John Wiley & Sons, New York, pp. 273–282.

HARMAN, M., C. FOX, R. HIERONS, L. HU, S. DANICIC, and J. WEGENER. 2002a. VADA: A transformation-based system for variable dependence analysis. *In* Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation. IEEE CS Press, Los Alamitos, CA, pp. 55–64.

HARMAN, M., L. HU, R. HIERONS, A. BARESEL, and H. STHAMER. 2002b. Improving evolutionary testing by flag removal. *In* Proceedings of the Genetic and Evolutionary Computation Conference, *Edited by* W. B. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska. Morgan Kaufmann, San Mateo, CA, pp. 1351–1358.

HENRION, M. 1988. Propagating uncertainty in Bayesian networks by probabilistic logic sampling. *In* Proceedings of the Second Conference on Uncertainty in Artificial Intelligence, *Edited by* J. F. Lemmer and L. N. Kanal. North-Holland, Amsterdam, pp. 149–163.

HERMADI, I., and M. A. AHMED. 2003. Genetic algorithm based test data generator. *In* Proceedings of the 2003 Congress on Evolutionary Computation, *Edited by* R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon. IEEE Press, Hoboken, NJ, pp. 85–91.

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

KULLBACK, S., and R. A. LEIBLER. 1951. On information and sufficiency. Annals of Mathematical Statistics, **22**:79–86.

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, *Edited by* C. Boutilier and M. Goldszmidt. Morgan Kaufmann, San Francisco, CA, pp. 343–352.

LARRAÑAGA, P., and J. A. LOZANO. 2002. Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation. Kluwer Academic Publishers, Boston.

LIN, J., and P. YEH. 2001. Automatic test data generation for path testing using GAs. Information Sciences, **131**:47–64.

LOZANO, J. A., P. LARRAÑAGA, I. INZA, and E. BENGOETXEA. 2006. Towards a new Evolutionary Computation: Advances in Estimation of Distribution Algorithms. Springer-Verlag, Dordrecht, The Netherlands.

MCGRAW, G., C. MICHAEL, and M. SCHATZ. 2001. Generating software test data by evolution. IEEE Transactions on Software Engineering, **27**(12):1085–1110.

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

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, *Edited by* H. M. Voigt, W. Ebeling, I. Rechenberger, and H. P. Schwefel. Springer-Verlag, London, pp. 178–187.

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

MÜHLENBEIN, H., and T. MAHNIG. 2001. Mathematical analysis of evolutionary algorithms for optimization. *In* Proceedings of the Third International Symposium on Adaptive Systems, La Havana, Cuba, pp. 166–185.

OCENASEK, J. 2006. Entropy-based convergence measurement in discrete estimation of distribution algorithms. *In* Towards a new Evolutionary Computation: Advances in Estimation of Distribution Algorithms, *Edited by* J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea. Springer-Verlag, Dordrecht, The Netherlands.

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. 2002. A survey of optimization by building and using probabilistic models. Computational Optimization and Applications, **21**(1):5–20.

PRESS, W. H., B. P. FLANNERY, S. A. TEUKOLSKY, and W. T. VETTERLING. 1988. Numerical Recipes in C. The Art of Scientific Computing. Cambridge University Press, New York.

SAGARNA, R., and J. A. LOZANO. 2005. On the performance of estimation of distribution algorithms applied to software testing. Applied Artificial Intelligence, **19**(5):457–489.

SAGARNA, R., and J. A. LOZANO. 2006. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. European Journal of Operational Research, **169**(2):392–412.

SAGARNA, R., and J. A. LOZANO. 2008. Software metrics mining to predict the performance of estimation of distribution algorithms in test data generation. *In* Knowledge-Driven Computing. Knowledge Engineering and Intelligent Computations, *Edited by* C. Cotta, S. Reich, R. Schaefer, and A. Ligeza, Springer, Berlin. In press.

SMITH, J., and T. C. FOGARTY. 1996. Evolving software test data—GA's learn self expression. *In* Proceedings of Evolutionary Computing. AISB Workshop, *Edited by* T. C. Fogarty. Springer-Verlag, Berlin, pp. 346–354.

SOMMERVILLE, I. 2001. Software Engineering. Addison-Wesley, Boston.

STHAMER, H. 1996. The Automatic generation of software test data using genetic algorithms. Ph.D. thesis. University of Glamorgan, Pontyprid, Wales, UK.

TRACEY, N. 2000. A search-based automated test-data generation framework for safety critical software. Ph.D. thesis. University of York, York, UK.

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, *Edited by* D. Redmiles and B. Nuseibeh. IEEE CS Press, Hoboken, NJ, pp. 285–288.

WEGENER, J., A. BARESEL, and H. STHAMER. 2001. Evolutionary test environment for automatic structural testing. Information and Software Technology, **43**(14):841–854.

WHITLEY, D., K. MATHIAS, and P. FITZFORN. 1991. Delta Coding: An iterative search strategy for genetic algorithms. *In* Proceedings of the 4th International Conference on Genetic Algorithms, *Edited by* R. K. Belew and L. B. Booker. Morgan Kaufmann, San Mateo, CA, pp. 77–84.