Available online at www.sciencedirect.com
SCIENCE DIRECT®
European Journal of Operational Research 169 (2006) 392–412
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms

Ramón Sagarna *, Jose A. Lozano

Department of Computer Science and Artificial Intelligence, University of the Basque Country, Paseo Manuel Lardizabal 1, 20018 San Sebastián, Guipuzcoa, Spain

Received 29 July 2003; accepted 22 January 2004

Abstract

One of the main tasks software testing includes is the generation of the test cases to be used during the test. Due to its expensive cost, the automatization of this task has become one of the key issues in the area. The field of Evolutionary Testing deals with this problem by means of metaheuristic search techniques.

An Evolutionary Testing based approach to the automatic generation of test inputs is presented. The approach developed involves different possibilities of the usage of two heuristic optimization methods, namely, Scatter Search and Estimation of Distribution Algorithms. The possibilities comprise pure Scatter Search options and Scatter Search—Estimation of Distribution Algorithm collaborations. Several experiments were conducted in order to evaluate and compare the approaches presented with those in the literature. The analysis of the experimental results raises interesting conclusions, showing these alternatives as a promising option to tackle this problem.

Keywords: Evolutionary computations; Software testing; Branch coverage; Scatter search; Estimation of Distribution Algorithms

1. Introduction

Testing plays a main role in the search for the required software quality. It is the primary way used in practice to verify the correct behaviour of the software produced. In fact, at least 50% of the project resources is usually dedicated to this phase (Beizer, 1990, 1995).

A key task in testing is the generation of the input cases to be applied to the programme under test. This is a critical task, as the input produced must fit in with the test type and its requirements. This, together with the fact that in most of the organizations generation is performed manually, results in a large amount of resources dedicated to the test data generation step. Thus, the

* Corresponding author.
E-mail addresses: ccbsaalr@si.ehu.es (R. Sagarna), ccploalj@si.ehu.es (J.A. Lozano).

0377-2217/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.ejor.2004.08.006

automatic generation of test cases has become one of the main lines of work in the area.

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

In contrast, structural testing is based on the internal structure of the programme. 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. These entities define several adequacy criteria which generators search to fulfil by producing adequate test inputs. For instance, branch coverage is a classical criterion that states that every programme branch must be exercised. Other typical criteria are passing through every code statement (statement coverage), exercising every sequence of branches from the programme input to its output (path coverage), or covering all the definition-usage pairs for each variable (all-defs coverage). Regardless of the adequacy criterion, there is a need to know the level of completion attained by the test case generator. This is enabled by the coverage measurement, i.e. the percentage of structural entities exercised for a given criterion.

In general, automated structural testing is reached by means of static or dynamic test data generation.

Static methods are commonly based on symbolic execution. This technique consists of choosing an entity from a graph representing the programme structure, and building a system of inequalities by assigning symbolic values to variables affecting the entity and respecting the constraints associated 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. Symbolic execution suffers from well-known problems, which limit its performance. The method requires a lot of computational resources, as expressions in the source code have to be resolved and transformed. In case a variable depends on a function call, no related inequality can be constructed if the source code of the function is unavailable. Other difficulties arise with array structures, pointers and loops (Korel, 1990).

These obstacles motivated the development of dynamic methods. The underlying idea in the dynamic alternative is addressing the automatic generation of test data as an optimization problem. An instrumented version of the programme is constructed, i.e. the programme is expanded with instructions that will extract information concerning the execution of an input. The collected information is used to assess the closeness of the executed inputs to cover the desired structural entities and guide the search towards new inputs to be executed. In Korel (1990), the obtained information determined a function value assigned to each input after execution. The objective was to find an input minimizing its function value, which only occurred when reaching the target entity.

A field which is arousing the interest of researchers is Evolutionary Testing (ET). The aim of ET is to seek test cases employing metaheuristic techniques during the process. In other words, ET searches for appropriate test inputs that are transformed into an optimization problem to be solved through these techniques. Promising results have already been attained in the field for structural adequacy criteria (Jones et al., 1996; Wegener et al., 2001; Sagarna and Lozano, 2003a).

The present work describes an ET approach for the automatic generation of test cases. The proposed method deals with the branch coverage adequacy criterion through a two-step process: selection of the next branch to exercise, and search for an input reaching this branch. This search step is dealt with by considering two evolutionary methods, namely, Scatter Search (SS) (Laguna and Martí, 2003) and Estimation of Distribution Algorithms (EDAs) (Larrañaga and Lozano, 2002).

SS is an emerging evolutionary search method which has been successfully applied to several difficult optimization problems (Campos et al., 2001). This technique is based on the maintenance of a low cardinality set of solutions, which is updated with new solutions obtained from the combination of the members of the set. SS has been compared with Genetic Algorithms (GAs) in permutation (Martí et al., in press) and function optimization problems (Laguna and Martí, 2000), producing high quality solutions in fewer evaluations than GAs.

On the other hand, EDAs are a new set of evolutionary algorithms which, instead of creating new individuals through the classical crossover and mutation operators, as occurs in GAs, estimate the probability distribution associated with the selected individuals and sample this distribution to create the next population. As EDAs have already been applied to the test case generation problem with excellent results (Sagarna and Lozano, 2003a), they constitute an adequate benchmark for comparison with SS. Moreover, they can be considered as a promising option from which SS may benefit in order to improve its performance.

The approach here exposed makes use of the optimization methods above in two different ways. On the one hand, pure SS alternatives are presented in order to evaluate their performance and compare them with the approaches in previous works. On the other hand, an SS–EDA collaborative approach is included. As far as we know, this is the first time that SS is applied to the test case generation problem, and the first time that SS and EDAs are combined.

The rest of the paper is organized as follows. The next section reviews ET and a few salient references from this field. Section 3 makes a general introduction to EDAs and briefly explains the EDA based test case generator used in this work. In Section 4 the SS methodology is presented, together with a description of the SS approach developed. Next, the experiments are described and their results analysed. Sections 6 and 7 explain the SS and EDA combining approach, and discuss the results of the conducted experiments. Finally, obtained conclusions and future work are included.

2. Evolutionary testing

Evolutionary Testing (ET) deals with software test data generation by means of metaheuristic search techniques. Typical examples of such techniques are Simulated Annealing (Kirkpatrick et al., 1983), Tabu Search (Glover and Laguna, 1997), and Genetic Algorithms (GAs) (Goldberg, 1989).

The manner in which the heuristic technique takes part remains open. However, the research on the field to date has basically concentrated on temporal behaviour testing and structural test data generation, whilst the usual heuristic technique has been the GA.

Temporal behaviour testing consists of verifying that execution timing constraints are satisfied. In Polheim and Wegener (1999), an approach relying entirely on a GA was presented. An individual represented an input given to the programme, and the fitness value was obtained by ranking the individuals execution times. The test ran until the GA reached a maximum number of generations or until an optimum was found, i.e. an input violated the timing constraints.

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

ET approaches following this idea are motivated by the fact that the space defined by inputs and goal is generally large and complex. As simple local searches perform poorly in such spaces, more sophisticated search techniques become a suitable alternative.

Thus, the test data generation is faced as the resolution of a number of function optimization problems. In order to achieve this, most of the works apply the two-step iterative process shown

Repeat until stopping criterion met
  $E \leftarrow$ Select entity to exercise
  Obtain input minimizing function for E

Fig. 1. Test case generation scheme.

in Fig. 1. In the first step, a previously identified structural entity is selected and marked as an objective, while, in the second step, the function optimization problem associated with the objective entity is solved.

Although different selection strategies exist, the common trend is to determine the objective entity with the help of a graph that reflects the structural characteristics of the programme. For example, in the case of branch coverage, a control flow graph (Fenton, 1985) is usually used. In this type of graph, a vertex x represents a basic block in the code, i.e. a maximal sequence of statements such that if one is executed then all do. There is an arc (x,y) if the control of the programme can be transferred from block x to y without crossing any other block. Thus, programme branches will come defined by every vertex x with outdegree$(x) > 1$.

The next phase of the scheme in Fig. 1 tackles an optimization problem: given the search space Ω formed by the programme inputs and a function $h : \Omega \to \mathbb{R}$, find $\boldsymbol{x}^* \in \Omega$ such that $h(\boldsymbol{x}^*) \leqslant h(\boldsymbol{x})$ $\forall\, \boldsymbol{x} \in \Omega$.

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

$$h(\boldsymbol{x}) = \begin{cases} l & \text{if } \mathbf{COND} \text{ not reached,} \\ d(\mathscr{A}_{\boldsymbol{x}}, \mathscr{B}_{\boldsymbol{x}}) + K & \text{if } \mathbf{COND} \text{ reached and} \\ & \quad \text{objective not attained,} \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

where l is the highest computable value, $\mathscr{A}_{\boldsymbol{x}}$ and $\mathscr{B}_{\boldsymbol{x}}$ are an appropriate representation of the values taken by $\mathscr{A}$ and $\mathscr{B}$ in the execution, $d(\mathscr{A}_{\boldsymbol{x}}, \mathscr{B}_{\boldsymbol{x}})$ is a distance measurement between $\mathscr{A}_{\boldsymbol{x}}$ and $\mathscr{B}_{\boldsymbol{x}}$, and $K > 0$ is a previously defined constant. Commonly, if $\mathscr{A}$ and $\mathscr{B}$ are numerical, then $\mathscr{A}_{\boldsymbol{x}}$ and $\mathscr{B}_{\boldsymbol{x}}$ are their values and $d(\mathscr{A}_{\boldsymbol{x}}, \mathscr{B}_{\boldsymbol{x}}) = |\mathscr{A}_{\boldsymbol{x}} - \mathscr{B}_{\boldsymbol{x}}|$. In the case of more complex data types, a binary representation of the values for $\mathscr{A}$ and $\mathscr{B}$ can be obtained and, for instance, let $d(\mathscr{A}_{\boldsymbol{x}}, \mathscr{B}_{\boldsymbol{x}})$ be the Hamming distance (Sthamer, 1996).

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

In Wegener et al. (2001), this type of objective function is smoothed by taking a level of proximity into account for the inputs that do not reach the conditional statement of the objective branch. In contrast, in the approach presented in Pargas et al. (1999) the function is only this proximity measurement. In Bottaci (2003), the objective function described above is analyzed and a number of alternatives are proposed to overcome the deficiencies found. A well-known drawback appears with flag conditions, e.g. if the comparison operator in the conditional expression is $\neq$. In this case, the function described above only takes three values and offers a poor grained search space. In order to solve this problem several possibilities based on code transformations are described in Harman et al. (2002) and Baresel and Sthamer (2003).

The number of works in the ET literature following the previous strategy is huge. In Michael and McGraw (2001), a GA is applied for condition/decision coverage of C/C++ programmes, that is, every branch needs to be taken and each expression in each conditional statement must be both true and false. In Lin and Yeh (2001), path coverage is sought by means of a GA. Once again, a GA is used in Jones et al. (1996) and Sthamer (1996) to deal with branch coverage, and Simulated Annealing is proposed in Tracey et al.

(1998). Recently, EDAs have successfully been applied for branch coverage (Sagarna and Lozano, 2003a), while in Sagarna and Lozano (2003b) SS was chosen as the optimization method. The Tabu Search technique has been used in the approach presented in Díaz et al. (2003). In Sagarna and Lozano (2003c), a promising alternative is presented. An EDA searches for the appropriate inputs in a search space, which may be modified during the test case generation process.

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

The present work deals with branch coverage for C/C++ programmes. As all the approaches described here share the same test case generation structure, its explanation in the following paragraphs remains valid throughout the rest of the work.

2.1. General framework

In order to deal with branch coverage, the general framework conforms to the scheme in Fig. 1. 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 state of 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 or unsuccessful treatment of every unexercised objective branch.

The optimization phase is tackled by means of a population based metaheuristic where a solution is a 0–1 string representing an input. The fitness value is determined by function h, which is based on the function type described in Eq. (1). If the minimum is found, then the branch is marked as covered and the represented input stored. Otherwise, the branch is marked as treated but uncovered.

Each branch is bound with a set of solutions (inputs) which is used as the seed population for the metaheuristic. This set is maintained in such a way that at every moment during the test case generation process the best solutions found for the branch are stored in its set. Thus, when solving the optimization problem of a branch b, besides evaluating a solution according to the function of b, it is evaluated for every other branch b'. Then, if the solution is better than the worst in the set associated with b', the latter is replaced by the new better solution. In this case, the quality of the set of branch b' has been increased, so if the branch had previously been treated, its state is marked as untreated. Fig. 2 shows the evaluation algorithm for a solution $\boldsymbol{i}$. The value of function h associated with branch b for a given input $\boldsymbol{x} \in \Omega$ is represented by $h_b(\boldsymbol{x})$.

Apropos the objective branch selection phase, a control flow graph is used to identify the branches at the initialization stage, and to help to decide the next branch to be selected during the process.

Candidate objective branches are those that are in the untreated state. The branch objective will be the one for which the mean objective function value in its associated set is best. In case there is a tie, a breadth first search is carried out, i.e. from the tied branches, selection of the one with the lowest level in the control flow graph. A pseudocode of the selection algorithm can be observed in Fig. 3.

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

$\boldsymbol{x} \leftarrow$ Translate solution $\boldsymbol{i}$ to input $\boldsymbol{x}$

Execute instrumented programme with $\boldsymbol{x}$

Repeat for each uncovered branch b

  $f_b^{\boldsymbol{i}} \leftarrow h_b(\boldsymbol{x})$

  $f_b^{\boldsymbol{w}} \leftarrow$ Find the evaluation of the worst solution $\boldsymbol{w}$ in the set associated with b

  If $f_b^{\boldsymbol{i}} < f_b^{\boldsymbol{x}}$

    Substitute $\boldsymbol{w}$ for $\boldsymbol{i}$ in the set associated with b

    If $f_b^{\boldsymbol{i}} = 0$

      Mark b as covered

    else

      Mark b as untreated

  If b is the objective

    $value \leftarrow f_b^{\boldsymbol{i}}$

Return $value$

Fig. 2. Pseudocode for the evaluation algorithm.

$\overline{f}_{best} \leftarrow \infty$

$objective \leftarrow \emptyset$

$tie \leftarrow false$

Repeat for each untreated branch b

  $\overline{f}_b \leftarrow$ Average function value in the set associated with b

  If $\overline{f}_b < \overline{f}_{best}$

    $\overline{f}_{best} \leftarrow \overline{f}_b$

    $objective \leftarrow b$

  If $\overline{f}_b = \overline{f}_{best}$

    $tie \leftarrow true$

If $tie = true$

  $objective \leftarrow$ Breadth first search between branches with $\overline{f}_{best}$ value

Return $objective$

Fig. 3. Pseudocode for the selection algorithm.

In order to illustrate how the test case generation can be performed, Fig. 4 presents an example function, written in C programming language, together with its associated control flow graph and its instrumented version. The reduced box on the right represents an output file of a hypothetical execution of the instrumented programme version.

The flow graph is used to select the next objective branch whose coverage will be pursued. Then, an optimization problem is raised, where a solution is a representation of the programme input, i.e. three integers in the example.

After an execution of the instrumented code, an output file contains the traversed basic block (numbered as in the control flow graph) at each line and, if the previous block had a conditional statement with an expression $\mathscr{A}\mathbf{OP}\mathscr{B}$, the values of $\mathscr{A}$ and $\mathscr{B}$. This information is then used to obtain the objective function value of the executed input (solution).

The instrumentation results shown by the output file of Fig. 4 correspond to the input $(1, 20, 31)$. Then, if the branch represented by arc

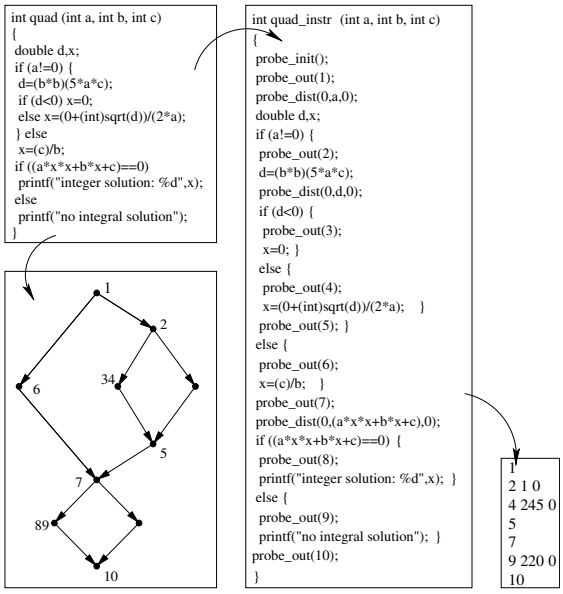

Fig. 4. Example function, control flow graph, instrumented version and output file.

(2, 3) in the graph is to be covered, the objective function value of the input is 246 (taking the absolute difference distance and $K = 1$ in Eq. (1)). Although the solution is already evaluated, the performed execution is not discarded and the output file is used to calculate the function value of every branch distinct to the present one.

3. Estimation of Distribution Algorithms

The EDA approach (Larrañaga and Lozano, 2002) comprises a set of emerging population based evolutionary algorithms firstly introduced in the field in Mühlenbein and Paaß (1996). In contrast to other evolutionary computation heuristics such as GAs, in EDAs there are neither crossover nor mutation operators. The individuals forming the population of the next generation are obtained sampling a probability distribution previously estimated from the selected individuals. Indeed, this probability distribution is responsible for one of the main characteristics of EDAs: the explicit description of the relationships between the variables defining individuals.

3.1. The abstract EDA

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

First, some notation which will be used in the rest of the section is required. Given an n-dimensional random variable $\boldsymbol{X} = (X_1, X_2, \ldots, X_n)$ 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 will be represented as $p(x_i|x_j) = p(X_i = x_i|X_j = x_j)$. In the context of evolutionary algorithms, an individual of length n is an instantiation $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ of $\boldsymbol{X} = (X_1, X_2, \ldots, X_n)$. The population of the lth generation, D_l, and the N selected individuals, 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}, i.e. the probability of an individual being among the selected individuals. This estimation must be carried out at each generation. Therefore, the joint probability distribution of the lth generation will be represented by $p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{\mathrm{Se}})$.

A pseudocode for the abstract EDA is presented in Fig. 5. The selection step can be carried out using any of the strategies in evolutionary computation, e.g. rank-based selection. Once the individuals are selected, they are treated as a database from which their probability distribution is estimated. Then this distribution is simulated to create the individuals of a new population. These steps are repeated until a previously defined stopping criterion is met.

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

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

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

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

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

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

Fig. 5. Pseudocode for abstract EDA.

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

Fig. 6 presents a schematic of an EDA approach. From the graph representing the variables' interdependencies, it is deduced that the estimation of the probability distribution is performed through a model, which allows a variable to depend on more than one other variable.

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

3.2. Univariate EDAs

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

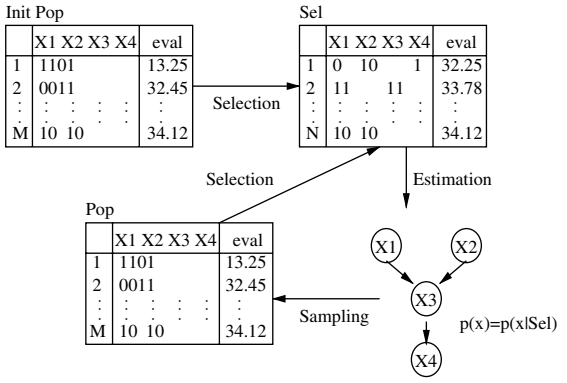

Fig. 6. Schematic of an EDA.

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

3.2.1. UMDA

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

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

where

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

3.3. Bivariate EDAs

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

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

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

3.3.1. TREE

TREE (Larrañaga and Lozano, 2002) refers to an adaptation of the Combining Optimizers with Mutual Information Trees (COMIT) algorithm (Baluja and Davies, 1997). The COMIT algorithm hybridizes the EDA approach with local search methods, while TREE only involves the EDA part. The estimation of the probability distribution is carried out using the Maximum Weight Spanning Tree (MWST) algorithm (Chow and Liu, 1968), which is based on the construction of a tree structure. Initially, the weight of each candidate branch (X_i, X_j) is calculated as the mutual information

measurement between X_i and X_j. Then, branches with the highest weight are increasingly added to the tree if they do not form a cycle in the graph. Once the structure is terminated, $p_l(\boldsymbol{x})$ is obtained through the conditional relative frequencies in D_{l-1}^{Se}.

Notice that in this model, two distinct variables may depend on the same one.

3.4. Multivariate EDAs

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

3.4.1. EBNA

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

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

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

In Fig. 7 an example of the structures associated with the TREE and EBNA algorithms are depicted. As can be seen, each variable in the TREE structure depends on one other variable at most, while in the case of EBNA a variable may depend on more than one variable.

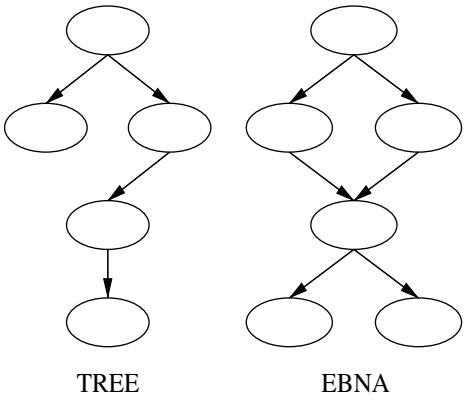

Fig. 7. Example of structures for EDAs.

3.5. EDAs in software testing

In Sagarna and Lozano (2003a) EDAs were used for the automatic generation of test cases to deal with branch coverage.

This approach followed the general framework described in Section 2 by tackling the optimization problem by means of an EDA. Thus, an individual was a 0–1 string representing an input, and its fitness was the value taken by the function h associated with the objective branch. The set of solutions associated with a branch was the same size as the population of individuals and, therefore, the initial population was this set. The EDA finished when either the minimum was found, i.e. an input covering the objective branch was found, or a maximum number of generations was reached.

4. Scatter Search

Scatter Search (SS) (Laguna and Martí, 2003; Glover et al., 2000) is an evolutionary search method whose principles were introduced in the early seventies. Due to the recent successful application of SS to several problems (Campos et al., 2001; Laguna and Martí, 2000), it is now one of the centres of attention of the combinatorial optimization community.

SS combines the solutions in a set, named *reference set*, in order to obtain new solutions to be added to the set. One of the main differences between SS and other population based evolutionary algorithms is the way selection of the solutions to be combined is performed. In SS, new solutions are created from two or more solutions chosen from the reference set according to a systematic strategy. In contrast, other evolutionary algorithms, such as GAs, obtain a new solution by random selection operators.

Initially, a P set of diverse solutions is constructed. The reference set, *RefSet*, is constructed by extracting adequate solutions from P, i.e. solutions with best objective function value or highest diversity in relation to the solutions already in *RefSet*. Then, a number of subsets of solutions is systematically generated. The solutions of these subsets will be combined to generate new solutions that may replace others in *RefSet*. Precisely, new solutions are improved with a local search method before considering their inclusion in *RefSet*. If a new solution has been added to *RefSet*, new subsets are generated and the process is repeated. Otherwise, the algorithm stops.

Structurally, the SS algorithm is composed of the following five interacting methods. The functionality of each method is clearly specified. However, its definition remains open to the problem being solved.

Diversification generation method. A method that generates a number of diverse solutions.
Improvement method. Once a solution is obtained, this method aims at improving it, usually through a local search method. Although this method is not strictly required, the common trend is to include it in the SS methodology.
Reference set update method. This method manages *RefSet* by defining the strategies necessary to build and update it. Both building and updating can take the objective function value or the diversity of the solutions into account. The basic SS algorithm stops when no new solution is added to *RefSet*. Nevertheless, in many cases a maximum number of iterations are established as the stopping criterion. Thus, if no solution is added to the set, a rebuilding step is carried out in the next iteration. This rebuilding consists of creating a new set P and replacing the half of worst solutions in *RefSet* with the solutions in P, which most increase the diversity in *RefSet*.
Subset generation method. The subsets of solutions are systematically generated from *RefSet*. At least, all subsets formed by two solutions are created. As the number of subsets tends to be high, there is a need for keeping *RefSet* small; generally, $|RefSet| = |P|/10$.
Solution combination method. This method creates new solutions by combining the solutions in a given subset.

The interaction of the five methods described above can be observed in the basic SS algorithm proposed in Fig. 8. A common size for P is 100 solutions and, therefore, $|RefSet| = 10$. Given that $|RefSet| = b_1 + b_2$, the reference set is usually constructed from the $b_1 = |RefSet|/2$ solutions in P with the best objective function value and the b_2 most diverse solutions. As noticeable, the subset generation method only considers the new subsets associated with the solutions introduced in the previous step. If the maximum number of iterations is not reached and no solution has been added to *RefSet*, then rebuilding is performed, i.e. P is created once again and b_2 of the solutions in *RefSet* are replaced.

Fig. 9 presents a schematic illustrating the roles of the SS methods. Circles represent new solutions, uncoloured before the application of the improvement method, and black afterwards. If no solution has been added to *RefSet*, the algorithm stops or makes a rebuilding step (dashed line), i.e. it executes the diversification method to obtain new diverse solutions.

4.1. SS in software testing

As pointed out in Section 2, when tackling the generation of a test input covering a branch, the associated search space is usually large and complex. A well-known conjecture in Operations Research is that an appropriate management of the diversification and intensification concepts during the search in such spaces yields to good solutions. These are the principles on which SS is based.

$P \leftarrow \emptyset$

$P \leftarrow$ Add $|P|$ distinct solutions obtained by diversification and improvement

$RefSet \leftarrow$ Add the b_1 solutions in P with best objective function value and delete them from P

Repeat for $l = 1, 2, ..., MaximumIteration$

  $RefSet \leftarrow$ Add the b_2 most diverse solutions in P in relation to the solutions in $RefSet$

  $NewSolutions \leftarrow TRUE$

  Repeat while $NewSolutions = TRUE$

    $NewSolutions \leftarrow FALSE$

    Generate all new subsets of solutions from $RefSet$

    Obtain new solutions by combination and improvement

    $RefSet \leftarrow$ Update $RefSet$ with new solutions

    If $RefSet$ changed then

      $NewSolutions \leftarrow TRUE$

    else

      $RefSet \leftarrow$ Delete the b_2 solutions with worst objective function value in $RefSet$

      $P \leftarrow \emptyset$

      $P \leftarrow$ Add $|P|$ distinct solutions obtained by diversification and improvement

Fig. 8. Pseudocode of basic SS.

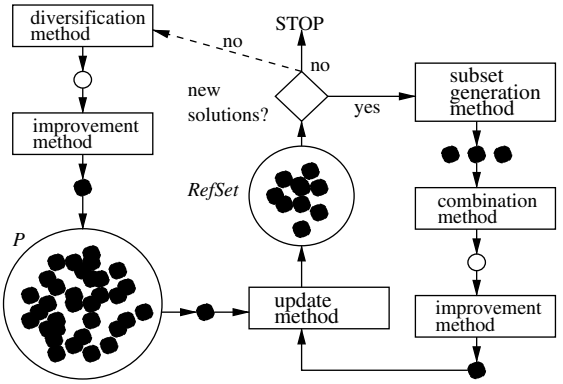

Fig. 9. Schematic of a basic SS design.

This, together with the flexibility of the SS methodology, makes it suitable for solving the test case generation problem. Moreover, in Laguna and Martí (2000) several SS designs were used to solve a set of nonlinear function minimization problems, obtaining encouraging results. Since the test case generation is tackled as the resolution of a number of function optimization problems, SS seems to be a promising technique to be studied.

The SS test case generator proposed here is based on the general framework described in Section 2. The optimization problem is tackled by means of an SS algorithm, where a solution is a 0–1 string representing an input and its objective function value is obtained according to Eq. (1). No advanced form of objective function was chosen in order to make a fair comparison with the EDA approach described in Sagarna and Lozano (2003a), which used a simple function.

The set of solutions of an objective branch, which has the same size as set P, is overturned in P at the beginning of each SS execution. Thus, in practice, the set of solutions can be viewed as a particular initial P set for each branch.

At the test case generation process start-up, the set of solutions of each branch b is created by introducing distinct solutions obtained via diversification and, if such is the case, via improvement. The evaluation of a solution is not only performed

in relation to b, but for any other branch b' with no completely constructed set. If the evaluated solution outperforms the worst in the set of b' and the solution is not in the set yet, then it is introduced. It is important to note that, in case an improvement method is used, only the best solution found for b' is considered for inclusion in its set, thus avoiding the introduction of solutions coming from the same seed. When starting the creation of the set of solutions for b, a number of them may already be in the set. However, they are not improved with regard to b because they were found during the improvement of another branch, so the improvement method is applied to these solutions. Nevertheless, no matter how many solutions are already in the set, the inclusion of half of them via diversification (and improvement) is forced in order to guarantee a degree of diversity in the set.

As may be noticed, once the process start-up finishes, every solution in the set of a branch is improved with regard to it. In order to maintain this property during test case generation, when a solution is evaluated and is to be included in the set of solutions of the branch, the improvement method is applied before entering the set. This way, at every moment the solutions in the set of a branch are improved with regard to it.

The SS stopping criterion consists of finding the minimum (covering the objective branch) or reaching a maximum number of iterations. If the current iteration is not the last and no new solution was added to *RefSet*, a rebuilding step is carried out.

None of the SS designs used in Laguna and Martí (2000) applied an improvement method, and this might be an important element during the search. Thus, in the present approach, with the purpose of shedding some light on how the use of the improvement affects the optimization process, the following options are given.

Improvement after. The classical way of improving the solutions, that is, after diversification or combination (Fig. 9).

Improvement before. An alternative consisting of using the improvement method just before entering *RefSet*. Once a solution has been created via diversification, it is included in P, and improvement is applied only if the solution is one of the b_1 high quality solutions used to construct *RefSet*. The other b_2 solutions of *RefSet* are not improved since they are assumed to be diverse solutions. Notice that in the rebuilding steps it is not necessary to improve the solutions from P. If a solution comes from the combination method, improvement is performed only if it is to enter *RefSet*.

Excepting those reaching the optimum, during an SS execution a number of solutions are obtained, improved and rejected if they do not gain entrance to *RefSet*. Therefore, the idea behind the Improvement Before alternative is to reduce the number of generated solutions by restricting improvement to those entering *RefSet*.

In order to complete the SS design description, the five methods needed to implement the SS algorithm are explained.

4.1.1. Diversification generation method

A simple implementation is adopted. Each solution is randomly generated according to a uniform distribution.

4.1.2. Improvement method

The improvement method is a best first local search where the neighbours of a solution are to a Hamming distance of one. More precisely, the bit substrings codifying each input parameter are taken into account to define the order of evaluation of the neighbours. For a solution differing from the previous one in the ith bit of the substring codifying the jth input parameter, the next neighbour to evaluate is obtained by changing the value of the most significant bit, previously unchanged, in the substring associated with the next parameter. To be exact, in the case of a parameter codification where the most significant bits are ordered from left to right, if the jth parameter is not the last, the ith bit of the substring belonging to the $(j+1)$th parameter will be flipped. Otherwise, the $(i+1)$th bit of the substring of the first parameter is changed.

Fig. 10 presents an example of the local search method. The solution to the left is the initial solution codifying an input with three parameters; a vertical line separates the substrings representing

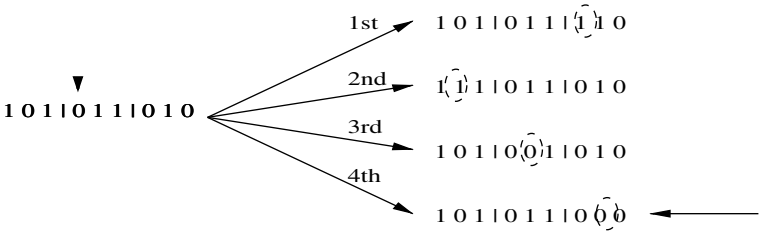

Fig. 10. Example of local search improvement method.

each parameter. It is supposed that this initial solution was obtained by changing the signed bit. The neighbours considered in a hypothetical search step are shown to the right. The horizontal arrow indicates the assumed new best solution chosen for the next step.

4.1.3. Reference set update method

The reference set updating follows a *static update* strategy. New solutions obtained via combination are placed in a pool. Once the pool is full, *RefSet* is formed by the highest quality solutions already in it and the pool.

4.1.4. Subset generation method

All two solution subsets are created. Obviously, only the solution pairs not generated previously are taken into account.

4.1.5. Solution combination method

For each pair of solutions, their test input representations $\boldsymbol{x}$ and $\boldsymbol{x}'$ are obtained, and four new solutions are created from the following linear combinations:

$$\boldsymbol{x}_1 = \boldsymbol{x} + d,$$
$$\boldsymbol{x}_2 = \boldsymbol{x} - d,$$
$$\boldsymbol{x}_3 = \boldsymbol{x}' + d,$$
$$\boldsymbol{x}_4 = \boldsymbol{x}' - d,$$

where $d = |\boldsymbol{x} - \boldsymbol{x}'|/2$.

5. Performance evaluation of Scatter Search designs

In order to observe how the SS approach performs in practice, several experiments were carried out. For this evaluation, a number of classical programmes in software testing experimentation were used. These are the following:

Triangle. This is a popular programme 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. Four different versions were used. The Triangle1 programme (Wegener et al., 2001) owns three integer parameters which, in the experiments, took values in the interval $[-16{,}384, 16{,}383]$; the number of branches (objectives) to be covered is 26. Triangle2 (Wegener et al., 2001) is the same as Triangle1 with floating point parameters instead, the interval for each was $[-98{,}304, 98{,}304]$. On the other hand, Triangle3 (Michael and McGraw, 2001) is a new implementation with integer parameters for which the interval $[-512, 511]$ was chosen; the number of branches is 20. Finally, Triangle4 (Sthamer, 1996) constitutes a distinct implementation once again, the selected interval for its integer parameters was $[-512, 511]$; 26 branches are included.

Atof. Given a string of characters as input, Atof (Wegener et al., 2001) transforms it into a floating point number if possible. For the experiments, the input string length was 10 characters codified with 7 bits each (the ASCII character set). 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 parameters for which the interval $[-32{,}768, 32{,}767]$ was chosen during experimentation. The Remainder programme has 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. Six integer parameters form an input taking values in the interval $[-512, 511]$; the source code reveals 22 branches.

Note that the number of branches in a programme is the number of optimization problems to be solved. However, this does not necessarily mean that the difficultness in generating test cases increases with the number of branches. The complexity in covering a branch is determined by its associated function, which is, in turn, defined by the corresponding conditional statement and the use of the variables in the programme.

After preliminary experimentation, the maximum number of generations for the SS was set at 10, the size of set P was 100 and the reference set size was 10. The results of the experiments for the two improvement strategies are shown in Table 1. In each cell, the average results from ten executions are provided. The first row is the average number of generated test inputs during the process, and the following is the average coverage measurement.

It can be seen in Table 1 that when the classical improvement strategy (Improve After) is adopted, the attained coverage is equal or larger than when Improve Before takes place. However, the latter offers good results, as it generates a considerably lower number of inputs (solutions) than Improve After. Indeed, in all programmes excepting Atof and Complexbranch, the coverage reached with the Improve Before option is the same or almost the same as the coverage reached with Improve After. These results indicate that the number of solutions that an SS algorithm generates during the optimization process depends, to a great extent, on the improvement method. Thus, although this is an optional element of SS methodology its relevance is high.

An interesting aspect which may be useful when considering an SS algorithm is the weight of each method during the search. This can be seen in Figs. 11 and 12 in the context of test case generation.

Fig. 11 shows the coverage attained by the SS methods for the Improvement After and Improvement Before strategies. Specifically, the results of the diversification, improvement and combination methods are presented, together with the coverage obtained when evaluating a solution with regard to other objectives. In the Improvement After alternative, almost all the coverage is reached through improvement and evaluation of other objectives. However, in the other alternative, where improvement is not so intense, the coverage of the local search decreases and the weight of diversification increases.

On the other side, Fig. 12 reveals the proportion of inputs generated by each SS method. Notice that, in general, improvement generates most of the solutions during the search; this is especially clear in the Improvement After option. An interesting point is the efficiency shown by diversification in the case of Improvement Before, since it generates a relatively low amount of inputs while offering most of the coverage.

A clear conclusion derives from these results. When used in a classical way, the improvement method plays a main role during the search, as it significantly affects the number and quality of generated solutions. In contrast, if improvement is applied differently, the behaviour of other SS methods changes, especially in connection with the achievement of high quality solutions.

Table 1
Experimental results of the SS approach

Improvement	Triangle1	Triangle2	Triangle3	Triangle4	Atof	Remainder	Complexbranch
After	10,304	19,357	27,007	36,129	1,504,311	141	38,984
	100	100	100	100	80	100	100
Before	1108	2166	5310	14,189	23,676	229	7228
	100	99.23	99	100	68	100	93.18

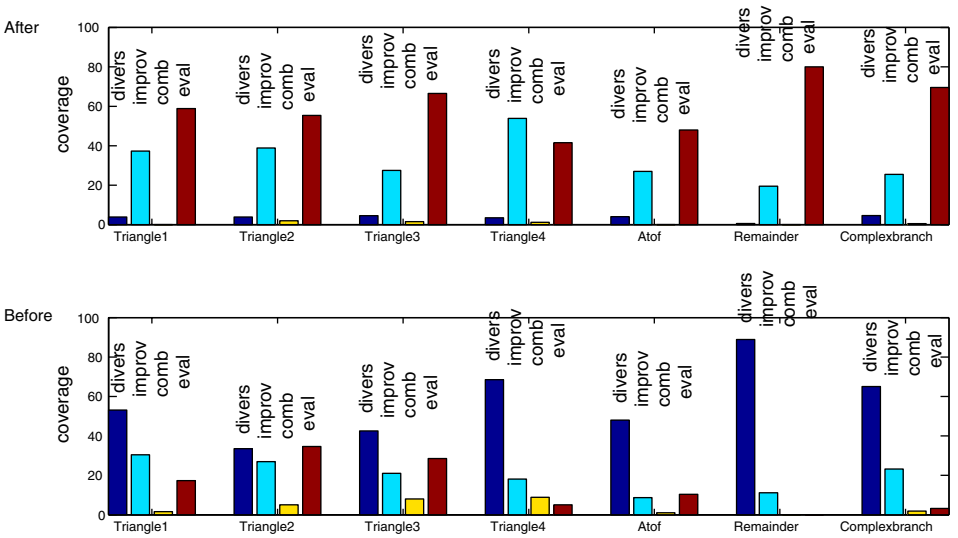

Fig. 11. Coverage of SS methods for Improvement After (above) and Improvement Before (below).

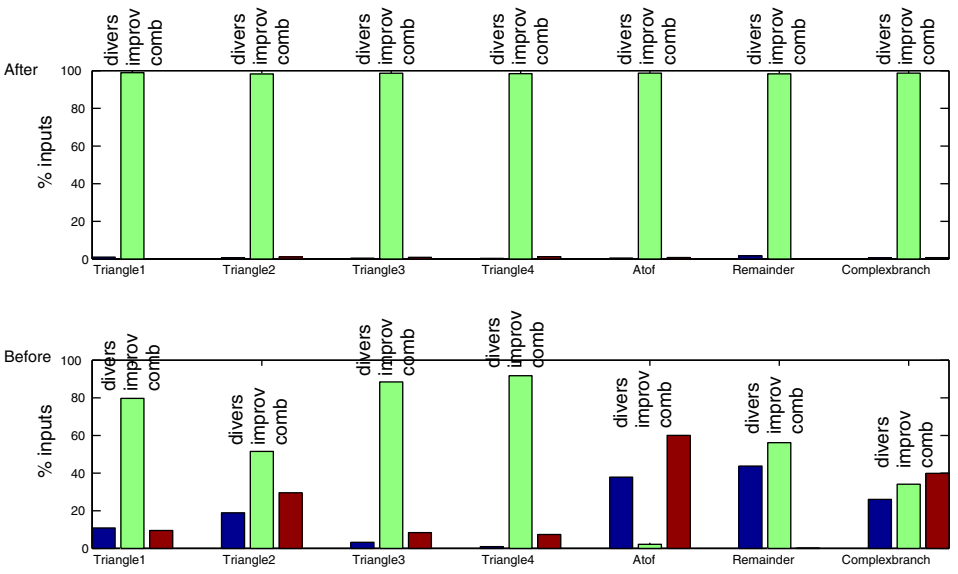

Fig. 12. Proportion of inputs generated by SS methods for Improvement After (above) and Improvement Before (below).

5.1. Scatter Search versus Estimation of Distribution Algorithms

In Sagarna and Lozano (2003a), EDAs were applied to the test case generation problem offering excellent results. Hence, the EDA approach may be regarded as an appropriate benchmark for comparison with the SS test case generator.

The programmes used in Sagarna and Lozano (2003a) for experimentation were the same as here. The EDAs applied comprised all the ones introduced in Section 3, and three more: Population

Table 2
Results of the best SS and EDA approaches

Approach	Triangle1	Triangle2	Triangle3	Triangle4	Atof	Remainder	Complexbranch
SS_{best}	1108	19,357	27,007	14,188	1,504,311	141	38,984
	100	100	100	100	80	100	100
EDA_{best}	6150	6200	3875	22,720	7685	2360	11,930
	100	100	100	100	100	100	100

Based Incremental Learning (Baluja, 1994), Mutual Information Maximization for Input Clustering (De Bonet et al., 1997), and $EBNA_{BIC}$ (Larrañaga et al., 2000).

For the comparison, the best EDA and SS approaches in each work are taken into account. These approaches are identified according to the following criterion.

Complete branch coverage is adopted as a mandatory requirement in some quality standards. Even so, the number of inputs obtained during the process reflects the effort made in the test case generation. Therefore, it is important for a generator to obtain a coverage value with the lowest cost, that is, producing as few inputs as possible. According to this, the best approach is be the one that achieves the highest coverage and, if there is a tie, the approach with the lowest number of generated inputs is selected.

Table 2 presents the results. SS_{best} and EDA_{best} denote the best SS and EDA approaches respectively; the format of each cell is the same as in Table 1.

In three of the seven programmes, SS outperforms EDA. More precisely, two of these SS approaches correspond to the Improvement Before strategy, which makes it an interesting option for test case generation. In contrast, although the Improvement After strategies equals the coverage of the EDAs (except Atof), the number of inputs generated is higher in all cases (except Remainder). This is significantly clear for the Atof programme, where SS_{best} offers a poor behaviour compared to EDA_{best}.

6. Scatter Search and Estimation of Distribution Algorithms collaboration

The results obtained in some of the experimental programmes suggest that SS approaches can generate good solutions with a low number of evaluations. Indeed, these results conform to those obtained in Martí et al. (in press), and Laguna and Martí (2000), where SS reached high quality solutions in fewer evaluations than GAs. Nevertheless, here, as well as in these works, it has been shown that there are functions for which the SS approach does not offer a good performance.

On the other side, as hinted in the previous section, EDAs were successfully applied to the automatic generation of test cases (Sagarna and Lozano, 2003a). However, in EDAs, it is difficult to set an explicit control of the diversification and intensification balance. In contrast, in SS this can be performed in a direct way due to its flexibility.

These observations motivated the idea of combining both optimization techniques. Both SS and EDA based approaches aim at generating test data for a given programme by themselves. However, they could be entirely used in order to deal with the same problem, thus leading to a collaborative approach, which may profit from the benefits of SS and EDAs.

The proposed collaboration consists of an EDA–SS approach where each search method acts separately. In other words, the EDA based generator is used first and, once it has finished, the SS based generator is employed over the remaining uncovered branches. This way, the general scheme in Fig. 1 is first applied with an EDA and, if it was not able to solve the complete problem, the scheme is repeated with SS.

An important feature of the implementation developed is that, after the execution of the EDA generator, the SS approach initializes the set of inputs of each uncovered branch with the set resulting from the EDA. Thus, the SS generator starts the search using the best solutions found by the

EDA. Although this seeding could involve a lack of diversity, the SS can recover from it through the diversification method in the rebuilding step.

7. Performance evaluation of collaborative approach

The collaborative approach was applied to the programmes described in Section 5 in order to observe their performance.

The EDAs given as examples in Section 3 were used in the experiments, i.e. UMDA, TREE and $EBNA_{K2+pen}$. Within an EDA, half of the population was selected at each generation according to a rank-based strategy. New individuals were simulated from the learnt probability distribution by means of Probabilistic Logic Sampling (Henrion, 1988), and the population was created in an elitist way. Population size was the same as the one for set P. Since the SS generator may be used after the EDA, the maximum number of generations was relaxed to 10. In fact, a few preliminary experiments were conducted and the best results corresponded to this value. In the EDA literature, other works that have obtained good results with low parameter values in the experiments exist (Inza et al., 2000).

The Improvement After option of the SS approach attained, in four of the programmes, a higher coverage value than the Improvement Before alternative. However, the latter clearly generated fewer inputs than the former. Therefore, the SS generator was evaluated taking both options into account. The SS parameter values were the same as in Section 5, $|P| = 100$, $|RefSet| = 10$, and 10 iterations at most.

Table 3 shows the results of the experiments; once again, the cell format is the same as in previous tables. The best results for each programme are marked in bold.

As can be seen, these results conform to the ones obtained in Table 1, since the Improvement After strategy reaches, in general, a higher coverage than the Improvement Before option. Instead, the latter generates a lower number of inputs than the former. In four programmes, Improvement After attains the best results. However, it may be noted that in almost all cases where 100% is achieved, the Improvement Before case generates fewer inputs than its Improvement After counterpart.

In order to have an idea of the behaviour of each generator in the collaborative scheme, Figs. 13 and 14 present the proportion of coverage and generated inputs in the EDA and SS methods respectively.

Fig. 13 reveals that the EDA based generator covers most of the objectives, since it operates first and can attain, among others, the easiest

Table 3
Results of EDA–SS approach with Improvement After (panel A) and Improvement Before (panel B)

EDA	Triangle1	Triangle2	Triangle3	Triangle4	Atof	Remainder	Complexbranch
Panel A							
UMDA	4015	**4250**	6281	59,615	**570,306**	6202	**24,154**
	100	**100**	100	100	**91.33**	100	**100**
TREE	**3272**	10,210	3870	39,838	795,693	6532	34,900
	100	100	100	99.62	86.67	100	100
$EBNA_{K2+pen}$	5185	7452	7369	41,283	1,066,775	4377	31,653
	100	100	100	99.62	83	100	100
Panel B							
UMDA	3311	4700	3927	34,850	28,278	3012	11,298
	100	98.85	99.5	99.62	79.67	100	96.36
TREE	3776	4857	**3439**	**14,259**	30,529	2614	9428
	100	98.85	**100**	**100**	78.67	100	96.82
$EBNA_{K2+pen}$	3317	4860	3548	11,437	33,620	**2197**	9189
	100	98.08	99	98.85	71.67	**100**	95

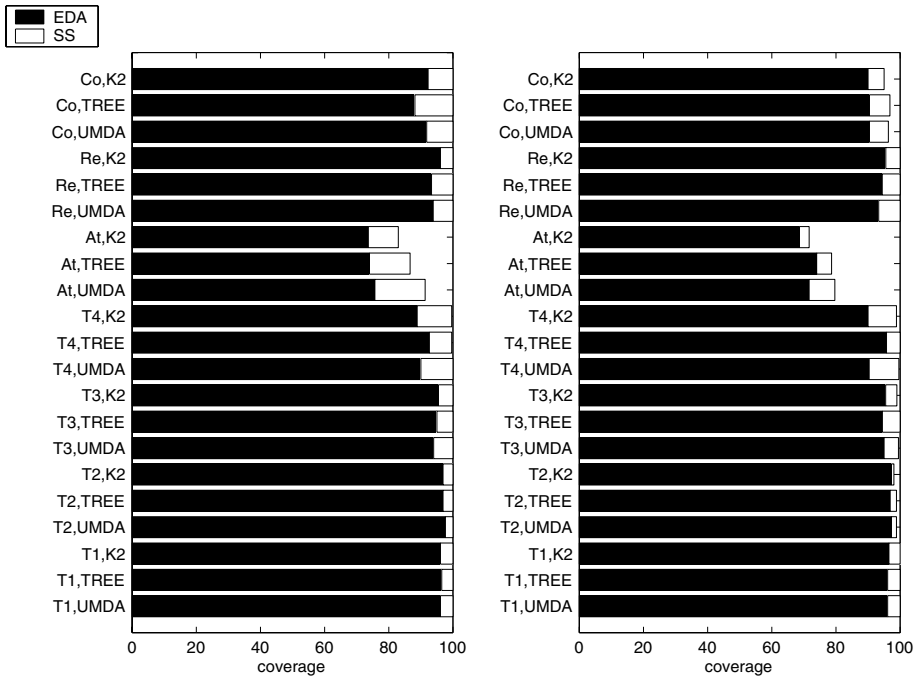

Fig. 13. Proportion in the coverage of EDA–SS approach for Improvement After (left) and Improvement Before (right).

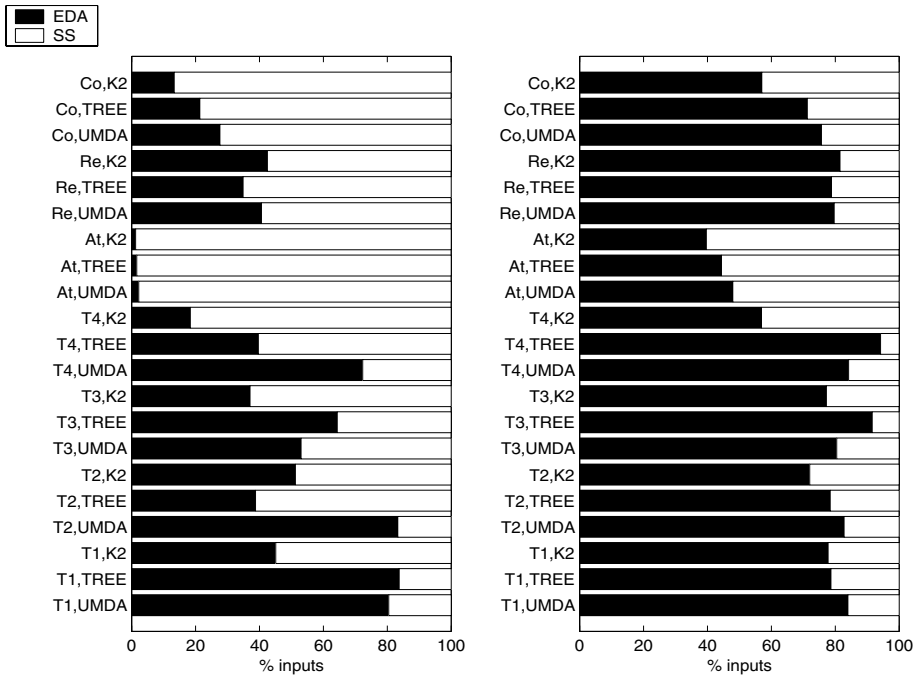

Fig. 14. Proportion of inputs of EDA–SS approach for Improvement After (left) and Improvement Before (right).

objectives. However, the EDA method is not able to reach a 100% coverage by itself, whilst this can be obtained by using the SS generator. In fact, the SS method always augments the coverage attained by the EDA based one. On the other hand, Fig. 14 shows how the use of improvement affects the results of the collaborative scheme, as the proportion of inputs generated by the SS with the Improvement After alternative is higher than with Improvement Before.

7.1. Collaborative approach versus others

The comparison of the EDA–SS approach with regard to the SS and EDA based test case generators can be observed in Table 4. In order to conform with the comparison in Table 2, the best EDA–SS collaboration ($EDA - SS_{\text{best}}$) is compared with the best SS (SS_{best}) and EDA (EDA_{best}) approaches.

Notice that $EDA - SS_{\text{best}}$ reaches better values than the others in most of the programmes. In four of them, the results of SS_{best} are improved, whilst in five programmes EDA_{best} is outperformed.

Similar to Section 5, a significant exception to these results is the poor performance attained in Atof. An explanation for this is that, even though preliminary experiments offered better results with the selected parameter values, in the case of Atof these values may not be appropriate. Nevertheless, Table 4 shows an increase in the coverage and a decrease in the inputs generated for $EDA - SS_{\text{best}}$ when compared to SS_{best}.

In any case, according to these results, the collaborative scheme is, at least, a competitive alternative for test case generation.

8. Conclusions

In this work, two new approaches for the automatic generation of test cases were described. On the one hand, in order to fulfil the branch coverage criterion, test inputs are searched by means of a SS generator. On the other hand, an EDA–SS collaborative scheme aims at taking advantage of the benefits of both methods. As far as we know, SS had never been previously used to solve this problem, and it is the first time EDAs and SS have been combined.

From the obtained experimental results, it may be concluded that, despite being optional, the improvement method plays a main role in the SS methodology. The weight of improvement is reflected in the number of solutions generated (inputs) and the number of optimums found during the search. On the other hand, the way in which improvement is used in the algorithm affects the behaviour of other SS methods. Following this idea, the Improvement Before option proposed in the SS approach attained better results than Improvement After in some programmes, thus being an interesting alternative to the classical strategy.

The experiments conducted on the SS and EDAs combination offered encouraging results. The collaborative approach outperformed the results of the EDA generator in five of the test programmes, and the results of the SS generator in four programmes. However, in order to conclude that the collaborative approach overcomes the isolated generators, future experiments have to confirm these results. Nevertheless, the use of SS as a secondary optimization method improved the coverage of the previous EDA based method.

Table 4
Results of best EDA–SS, SS and EDA approaches

Approach	Triangle1	Triangle2	Triangle3	Triangle4	Atof	Remainder	Complexbranch
$EDA - SS_{\text{best}}$	3272	4250	3439	14,259	570,306	2197	24,154
	100	100	100	100	91.33	100	100
SS_{best}	1108	19,357	27,007	14,188	1,504,311	141	38,984
	100	100	100	100	80	100	100
EDA_{best}	6150	6200	3875	22,720	7685	2360	11,930
	100	100	100	100	100	100	100

Hence, the collaborative strategy proves to be useful.

Future work should be addressed towards the effect of each SS method during the search process. This would help in the design of the appropriate SS algorithm for a given optimization problem and, more precisely, in the case of test case generation. In addition, the adequate balance between the parameter values of each generator in the collaborative approach should be studied in order to obtain the maximum benefit. Finally, more forms of the collaborative scheme should be considered, e.g. SS–EDA combination, since it presents itself as a powerful option to tackle this problem.

Acknowledgments

This work was supported by the Ministry of Science and Technology under grant TIC2001-2973-C05-03, as well as by the Etortek-Genmodis and Etortek-Biolan projects of the Basque Government. R. Sagarna was also supported by the Department of Education, Universities and Research of the Basque Government within the programme of Researcher Education.

References

Baluja, S., 1994. Population based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh.

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

Baresel, A., Sthamer, H., 2003. Evolutionary testing of flag conditions. In: Proceedings of the Genetic and Evolutionary Computation Conference. Springer, pp. 2442–2454.

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

Beizer, B., 1995. Black-Box Testing Techniques for Functional Testing of Software and Systems. John Wiley & Sons.

Bottaci, L., 2003. Predicate expression cost functions to guide evolutionary search for test data. In: Proceedings of the Genetic and Evolutionary Computation Conference. Springer, pp. 2455–2464.

Burton, S., 2000. Automated testing from Z specifications. Technical Report YCS329, Department of Computer Science, University of York, Heslington, York.

Campos, V., Glover, F., Laguna, M., Martí, R., 2001. An experimental evaluation of a scatter search for the linear ordering problem. Journal of Global Optimization 21, 397–414.

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

Chow, C., Liu, C., 1968. Approximating discrete probability distributions with dependence trees. IEEE Transactions on Information Theory 14, 462–467.

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

De Bonet, J.S., Isbell, C.L., Viola, P., 1997. MIMIC: Finding optima by estimating probability densities. In: Advances in Neural Information Processing Systems 9. MIT Press, pp. 424–430.

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

Díaz, E., Tuya, J., Blanco, R., 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 Press, pp. 310–313.

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

Fenton, N.E., 1985. The structural complexity of flowgraphs. In: Alavi, Y., Chartrand, G., Lesniak, L., Lick, D.R., Wall, C.E. (Eds.), Graph Theory with Applications to Algorithms and Computer Science. John Wiley & Sons, pp. 273–282.

Glover, F., Laguna, M., 1997. Tabu Search. Kluwer Academic Publishers.

Glover, F., Laguna, M., Martí, R., 2000. Fundamentals of scatter search and path relinking. Control and Cybernetics 39 (3), 653–684.

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

Harman, M., Hu, L., Hierons, R., Baresel, A., Sthamer, H., 2002. Improving evolutionary testing by flag removal. In: Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers, 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. North-Holland, pp. 149–163.

Inza, I., Larrañaga, P., Etxeberria, R., Sierra, B., 2000. Feature subset selection by Bayesian networks based optimization. Artificial Intelligence 123 (1–2), 157–184.

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

Kirkpatrick, S., Gellat, C.D., Vecchi, M.P., 1983. Optimization by simulated annealing. Science 220, 671–680.

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

Laguna, M., Martí, R., 2000. Experimental testing of advanced scatter search designs for global optimization of multimodal functions. Technical Report, University of Colorado at Boulder, Boulder.

Laguna, M., Martí, R., 2003. Scatter Search. Methodology and Implementations in C. Kluwer Academic Publishers.

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

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

Lin, J., Yeh, P., 2001. Automatic test data generation for path testing using GAs. Information Sciences 131, 47–64.

Martí, R., Laguna, M., Campos, V., in press. Scatter Search vs. Genetic Algorithms: An experimental evaluation with permutation problems. In: Rego, C., Alidaee, B. (Eds.), Adaptive Memory and Evolution: Tabu Search and Scatter Search, Kluwer.

Michael, C.C., McGraw, G., 2001. Generating software test data by evolution. IEEE Transactions on Software Engineering 27 (12), 1085–1110.

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

Mühlenbein, H., Paaß, G., 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. Springer, pp. 178–187.

Pargas, R., Harrold, M., Peck, R., 1999. Test-data generation using genetic algorithms. Journal of Software Testing, Verification and Reliability 9 (4), 263–282.

Pelikan, M., Goldberg, D.E., Lobo, F., 1999. A survey of optimization by building and using probabilistic models. Computational Optimization and Applications 21 (1), 5–20.

Polheim, H., Wegener, J., 1999. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In: Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E. (Eds.), Proceedings of the Second Genetic and Evolutionary Computation Conference. Morgan Kaufmann, pp. 1795–1802.

Roper, M., Maclean, I., Brooks, A., Miller, J., Wood, M., 1995. Genetic algorithms and the automatic generation of test data. Technical Report EFoCS-19-95, University of Strathclyde, Glasgow.

Sagarna, R., Lozano, J.A., 2003a. On the performance of estimation of distribution algorithms applied to software testing. Technical Report EHU-KZAA-IK-1/03, The University of the Basque Country, Spain.

Sagarna, R., Lozano, J.A., 2003b. Scatter search and estimation of distribution algorithms in software testing: Competition and cooperation. Technical Report EHU-KZAA-IK-2/03, The University of the Basque Country, Spain.

Sagarna, R., Lozano, J.A., 2003c. Variable search space for software testing. In: Proceedings of the IEEE International Conference on Neural Networks and Signal Processing. IEEE Press, pp. 575–578.

Smith, J., Fogarty, T.C., 1996. Evolving software test data—GA's learn self expression. In: Fogarty, T.C. (Ed.), Proceedings of Evolutionary Computing. AISB Workshop. Springer-Verlag, pp. 346–354.

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

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

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