An Evaluation of Differential Evolution in Software Test Data Generation

R. Landa Becerra, R. Sagarna and X. Yao, *Fellow, IEEE*

***Abstract*— One of the main tasks software testing involves is the generation of the test inputs to be used during the test. Due to its expensive cost, the automation of this task has become one of the key issues in the area. Recently, this generation has been explicitly formulated as the resolution of a set of constrained optimisation problems. Differential Evolution (DE) is a population based evolutionary algorithm which has been successfully applied in a number of domains, including constrained optimisation. We present a test data generator employing DE to solve each of the constrained optimisation problems, and empirically evaluate its performance for several DE models. With the aim of comparing this technique with other approaches, we extend the experiments to the Breeder Genetic Algorithm and face it to DE, and compare different test data generators in the literature with the DE approach. The results present DE as a promising solution technique for this real-world problem.**

I. Introduction

Testing is the primary way used in practice to verify the correctness of software [1]. Among the problems related to software testing, the automatic generation of the inputs to be applied to the programme under test is especially relevant. Exhaustive testing is generally prohibitive due to the huge size of the input domain, so tests are designed with the purpose of fulfilling particular adequacy criteria. For instance, *branch coverage* is accepted as a minimum mandatory criterion [1]. So, in this case, the aim is to generate a set of inputs exercising every branch in the source code of the programme.

In recent years, several approaches under the name of Search Based Software Test Data Generation (SBSTDG) have been developed, offering promising results [2]. SBSTDG tackles the test data generation as a search for the appropriate inputs by formulating an optimisation problem. This problem is then addressed using search methods. Most of the SBSTDG approaches follow a dynamic strategy where the programme is executed and the information available at run-time is exploited to guide the search of test inputs [3], [4], [5], [6].

Recently, dynamic SBSTDG for branch coverage has been explicitly formulated as the resolution of a set of constrained optimisation problems [7]. This formulation opens the door to new designs and search strategies that have not been considered to solve this problem yet. In this work, we focus on the application of Differential Evolution (DE) [8].

R. Landa Becerra, R. Sagarna and X. Yao are with The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK; email: {R.B.Landa,R.Sagarna,X.Yao}@cs.bham.ac.uk .

Considering the search strategy for this approach, we looked for one to be successful on constrained optimisation tasks, but without consuming a large number of fitness function evaluations (which here means programme runs). An alternative fulfilling such conditions is DE. DE has shown its effectiveness in a number of benchmark functions [9] and real-world problems [10], and it has successfully been applied in constrained optimisation, obtaining good solutions and also reducing the number of function evaluations [11], [12], [13], [14]. DE is a population based evolutionary algorithm which is currently deserving the attention of the research community. A major characteristic of this algorithm is that new individuals are obtained by combining one parent and a trial individual elicited from the vector differential of two other individuals. Another important feature is that selection is local, in the sense that only the parent is considered for replacement, which results in a inherently high selection pressure.

We build a test data generator which employs DE to solve the constrained optimisation problems associated to branch coverage. We then evaluate the performance of different DE models through experiments over a set of five benchmark programmes. To the best of our knowledge, this is the first application of DE to software testing following the mentioned constraints based formulation. In order to have an idea of the performance of this technique compared to other population based evolutionary algorithms, we also experiment with the Breeder Genetic Algorithm [15], which is a well-known technique with a wide track record of successful real-world applications [16], [17]. Also, we face the DE based generator with other SBSTDG approaches from the literature [6], [5], [18], [4], [19], [7].

The remaining sections are arranged as follows. In the next section, dynamic SBSTDG and the constrained optimisation formulation are outlined. Then, the standard DE algorithm and some of its variants are briefly reviewed. We continue with the empirical evaluation of different DE models. Finally, we discuss conclusions and some ideas for future work.

II. Dynamic Search Based Software Test Data Generation

SBSTDG methods obtain test inputs employing search techniques during the process. The main characteristic of dynamic strategies is that they execute an instrumented version of the programme at hand with an input. The information gathered during the run time is then used to guide the search for new inputs. Next, we first discuss classical dynamic

SBSTDG for branch coverage and afterwards the constrained optimisation approach we have followed.

A. Classical Approaches

Although different works have been developed in the field to date (see [2] and references therein), the idea underlying many of them is to solve a number of function optimisation problems, one for each branch to be covered. Thus, it is common to follow a two-step iterative process where, firstly, a branch is selected and marked as an objective, and secondly, this objective is assigned a function dependent on the programme input and its optimisation is sought.

1) Selection Step: The objective branch is often determined with the help of a *control flow graph* [20] which reflects the structural characteristics of the programme [3], [6].

A control flow graph $G = (V, U)$ is defined by a set V of vertices and a set $U \subseteq V \times V$ of arcs. Each vertex in V represents a *code basic block*, excepting two vertices labelled s and e, which refer to the programme entry and exit. A code basic block is a maximal sequence of code statements such that if one is executed, then all of them are. An arc $(v_1, v_2) \in U$, with v_1 and v_2 distinct from s and e, is such that the control of the programme can be transferred from block v_1 to v_2 without crossing any other block. Analogously, for every arc $(s, v_1) \in U$ or $(v_2, e) \in U$, it will be possible to transfer the flow of control from the entry to block v_1 and from block v_2 to the exit, respectively. Hence, in this kind of graph, every vertex v with $outdegree(v) > 1$ represents a branch in the source code of the programme. Given a programme input $\boldsymbol{x}$, we will call *execution path of* $\boldsymbol{x}$ to the path starting from s that represents the flow of the programme's control when executed with $\boldsymbol{x}$.

2) Optimisation Step: In this step an optimisation problem is tackled. That is, given the search space Ω formed by the programme inputs and a function $f : \Omega \longrightarrow \mathbb{R}$, find $\boldsymbol{x}^* \in \Omega$ such that $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})\ \forall \boldsymbol{x} \in \Omega$.

A measure that is widely used to create f is the so-called *branch distance* [2]. Let b be the objective branch and $\mathcal{A}\,\mathbf{OP}\,\mathcal{B}$ an expression of the conditional statement $\mathbf{COND}$ associated with b in the code, with $\mathbf{OP}$ denoting a comparison operator. Only for notation purposes, we also consider the vertex v_c representing $\mathbf{COND}$ in the control flow graph of the programme. The branch distance value for an input $\boldsymbol{x}$ that reaches $\mathbf{COND}$ is determined by

$$f^c(\boldsymbol{x}) = d(\mathcal{A}_{\boldsymbol{x}}, \mathcal{B}_{\boldsymbol{x}}) + K \tag{1}$$

where $\mathcal{A}_{\boldsymbol{x}}$ 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, other representations and distances have been proposed in the literature [5]. Besides, if $\mathbf{COND}$ involves a compound expression, the overall branch distance can be obtained by combining the distances at the subexpressions [6].

A classical objective function based on the branch distance is defined, keeping the notation above, as follows [3]:

$$f(\boldsymbol{x}) = \begin{cases} L & \text{if } \mathbf{COND} \text{ not reached} \\ f^c(\boldsymbol{x}) & \text{if } \mathbf{COND} \text{ reached and } b \text{ not attained} \\ 0 & \text{otherwise} \end{cases}$$

where L is the largest computable value.

In [6], the gradient of this function was increased for the inputs not reaching $\mathbf{COND}$ by defining a distance between a vertex in the execution path of the input and $\mathbf{COND}$.

3) Other Elements of a Test Data Generator: Although the search technique deals with one optimisation problem at a time, the real goal is to solve a set of problems. Several works have taken this into account to improve the process. The alternative suggested by them is to profit from the good solutions found by not only evaluating an input for the current objective branch, but also with regard to all the others. Thus, each branch is assigned a set containing the best inputs so far. The strategy to select the objective branch consists then of choosing the branch with a highest quality set of inputs. Moreover, for the optimisation step, this set is used to seed the initial phase of the search method [3], [4], [6]. This way, at each round, we try solve the optimisation problem with the most promising initial solutions for the search technique.

B. Dynamic SBSTDG as Constrained Optimisation

Recently, dynamic SBSTDG for branch coverage has been explicitly formulated as a constrained optimisation problem [7], so opening the door to techniques not considered so far. This formulation is based on the notion of *critical condition*.

Given a control flow graph, we call a vertex v_1 a critical condition of vertex v_2 iff $outdegree(v_1) > 1$, a path from v_1 to v_2 exists, and a path from v_1 to e not containing any vertex in any path from v_1 to v_2 exists. Intuitively, a critical condition v_1 has an arc from which it is impossible to attain v_2, so we must follow one of the other arcs. For instance, in the control flow graph shown in Figure 1, v_1, v_2 and v_4 are critical conditions of v_c, but v_3 is not.

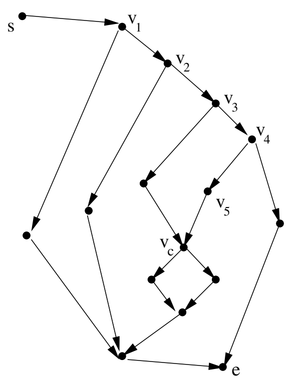

Fig. 1. Example of a control flow graph.

The attainment of the objective branch, represented by arc (v_c, v_o) in the control flow graph $G = (V, U)$, consists of finding an input whose execution path contains (v_c, v_o).

Although several such paths may be possible, the critical conditions of v_c indicate the arcs we must follow to achieve v_c, i.e. they identify a set of arcs that are common to different paths from s to v_c. We will call this set of arcs a *critical set for* v_c. For example, in Figure 1, $\{(v_1, v_2), (v_2, v_3), (v_4, v_5)\}$ is a critical set for v_c.

Let $\zeta = \{(v_i, v_i'),\ v_i, v_i' \in V,\ \forall i \in \{1, 2, ..., n\}\}$ be a critical set for v_c. The coverage of each branch b_i, represented by $(v_i, v_i') \in \zeta$, is then a constraint we must fulfil to attain the objective branch. The constraint function for b_i is given by

$$g^i(\boldsymbol{x}) = \begin{cases} \frac{f^i(\boldsymbol{x})}{M} & \text{if } b_i \text{ not attained} \\ 0 & \text{otherwise} \end{cases}$$

where f^i is the branch distance in Equation 1 and M is a normalisation term.

Now, we can formulate the coverage of the objective branch b as a constrained optimisation problem (we keep the notation)

$$\begin{array}{l} minimise \quad f(\boldsymbol{x}) = \begin{cases} \frac{f^c(\boldsymbol{x})}{M} & \text{if } b \text{ not attained} \\ 0 & \text{otherwise} \end{cases} \\ s.t. \quad g^i(\boldsymbol{x}) = 0, \quad i = 1, 2, ..., n. \end{array} \qquad (2)$$

Following this formulation the test data generation is transformed into the resolution of a set of constrained optimisation problems. This view can lead to new designs for the components of the generator.

1) Selection Step: It is important to notice that different critical sets might exist for v_c, e.g., in Figure 1, $\{(v_1, v_2), (v_2, v_3), (v_4, v_5)\}$ and $\{(v_1, v_2), (v_2, v_3)\}$ are both critical sets for v_c. This implies several constrained optimisation problems might exist for the same objective branch.

In order to maximise the number of branches covered when the objective branch is attained, the strategy proposed in [7] is to choose as objective the branch with the largest critical set. The constrained optimisation problem is then given by this set. In case of tie among branches, the one with a highest quality set of inputs is selected (i.e., the strategy described in II-A.3 is adopted). The quality of a set is taken to be the mean objective function value of the inputs in the set. Analogously, if different critical sets with equal cardinality exist for the same branch, we can keep one set of inputs for each of them. Then, we can choose the critical set with the highest quality set of inputs associated.

2) Optimisation Step: A large amount of classical SB-STDG generators [3], [4], [5], [6] implicitly conform to a constraint-handling approach [7]. However, all of them handle the constraints in the order naturally imposed by this problem: the value of $g^i(\boldsymbol{x})$ is unknown unless $g^{i-1}(\boldsymbol{x}) = 0$. Therefore, the search points are encouraged to pursue the optimal regions defined by constraints in this particular order. Depending on the topology of these regions and the functions encoded by the constraints, this demarcation of the path to the optimum may hinder the search. Additionally, such a restrictive way of achieving each constraint might lead to a lack of diversity [21].

If we were able to overcome the restriction of following the order naturally given by the problem we would dramatically open the range of search techniques that can be applied to the test data generator.

Actually, this can be achieved through the *testability transformation* presented in [22]. Such a transformation is a controlled modification of the source code of the programme which aims at improving some aspect of a test data generator. The testability transformation proposed in [22] consists of a particular instrumentation of the source code that allows to obtain the branch distance value for every critical condition associated to the objective branch. The main idea to achieve this is to remove the conditional statements corresponding to the critical conditions of the objective branch, and to compute the branch distance instead. This way, we can calculate the value for any constraint g^i and for the objective function f in Equation 2. Figure 2 gives an example of the type of instrumentation presented in [22]. Only one critical condition is considered, which corresponds to the branch distance f^2.

It is worth to remark, however, that this instrumentation might pose some issues for certain conditional statements. For instance, it might be the case the branch distance is not defined (has no value) for the input at hand. In this situation, we could choose to return the worst possible value for that condition. Further issues are discussed in [22].

Having a means to calculate any of the values in Equation 2 for any programme input, we can use the techniques developed in the field of constrained optimisation. However, the general problem of meeting a set of constraints is known to be NP-complete [23], which has motivated the widespread use of approximation algorithms. Since virtually any function may be encoded in a condition and, hence, in the branch distance (Equation 1), we may assume the same complexity for the general case of branch coverage when following the formulation in Equation 2. In the next sections, we concentrate on the evaluation of Differential Evolution to solve this problem.

III. Differential Evolution

Differential Evolution (DE) is a recently developed evolutionary algorithm originally proposed by Price and Storn [8], whose main design emphasis is real parameter optimisation, but that has been applied successfully to mixed integer problems [10]. DE is based on a mutation operator, which adds an amount obtained by the difference of two individuals randomly chosen from the current population, in contrast with most evolutionary algorithms, in which mutation is performed through a random variable.

The basic algorithm is shown in Algorithm 1, which presents the most widely adopted variant of DE; $x_{i,j}$ is the i-th variable of the j-th individual in the population, F and CR are parameters given by the user (called difference and recombination constants, respectively), and $U(a, b)$ is a realisation of a uniformly distributed random variable between a and b. The function $\text{isbetter}(\boldsymbol{x}_a, \boldsymbol{x}_b)$ returns `true` if $\boldsymbol{x}_a$ is better than $\boldsymbol{x}_b$, considering a given criterion (commonly, the value of the objective function).

```
   void example (int a, int b)
(1){
(2)  if (a<b)
(3)    if (a*a-b+5==0)
(4)      // objective branch
(5)}
```

```
   void example_transformed (int a, int b)
(1){
(2)  compute_bd(a<b);            ← returns f^2(a,b)
(3)  compute_bd(a*a-b+5==0);     ← returns f^3(a,b)
(4)  if (a*a-b+5==0)
(5)    // objective branch
(6)}
```

Fig. 2. Example of the type of instrumentation proposed in [22]. Original programme to the left and transformed programme to the right.

Algorithm 1 Differential Evolution (DE/$rand$/1/bin version)

```
initialise($P = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\}$)
evaluate($P$)
repeat
  for $j = 0$ to $M$ do
    Let $r_1$, $r_2$ and $r_3$ be three random integers in $[1, M]$, with $r_1 \neq r_2 \neq r_3$
    Let $i_{\mathrm{rand}}$ be a random integer in $(1, n)$
    for $i = 1$ to $n$ do
      $x'_{i,j} = \begin{cases} x_{i,r_3} + F \cdot (x_{i,r1} - x_{i,r2}) & \text{if } U(0,1) < CR \text{ or } i = i_{rand} \\ x_{i,j} & \text{otherwise} \end{cases}$
    end for
    evaluate($\boldsymbol{x}'_j$)
    if isbetter($\boldsymbol{x}'_j, \boldsymbol{x}_j$) then
      $\boldsymbol{x}_j = \boldsymbol{x}'_j$
    end if
  end for
until the termination condition is achieved
```

The algorithm's main component is the variation operator. Another important part of DE is the selection operator, which performs the selection progressively during the generation of children, making only local comparisons. The function isbetter is adopted to test the condition for replacement of the parent $\boldsymbol{x}_j$ for the child $\boldsymbol{x}'_j$. These local comparisons make the algorithm more efficient, avoiding the need of sorting or ranking the population.

The DE approach has been adopted several times to solve mixed integer and continuous parameter problems, with encouraging results (for example, in [10]). The modification most frequently used is very simple, and it do not interfere with the algorithm's internal functioning. Such modification is made by truncating the real parameter before evaluating the individual. It is important to mention that the upper bound for domain constrained variables must be extended to the next integer. That is, if the original integer variable i has domains intervals $[l_i, u_i]$, then the bounds for the continuous variable must be $[l_i, u_i + 1)$.

A. Differential Evolution Variants

As mentioned above, Algorithm 1 shows the variant called rand/1/bin, which is the most commonly adopted. There exist other variants of differential evolution, some with better performance in certain domains [24]. However, two approaches appear more frequently in literature, perhaps because they are useful in a wide range of problems [9], [25], [24]. They are DE/$rand$/1/bin and DE/$best$/1/bin (for details on this notation, please refer to [9]).

To obtain the DE/$best$/1/bin variant, only a simple change is needed in the mutation step of the algorithm. Change the differential expression $x_{i,r_3} + F \cdot (x_{i,r1} - x_{i,r2})$ in Algorithm 1 for $x_{i,\mathrm{best}} + F \cdot (x_{i,r1} - x_{i,r2})$ where $x_{i,\mathrm{best}}$ is the i-th variable of the best individual found so far. This variant is more greedy, but it can accelerate convergence, and provide good results if the search space is not too complex.

Other notable modifications proposed for the DE, are regarding the parameter F. Some authors have noted that assigning the F a random value may improve the convergence [26]; this process, called *dithering*, is usually regarded as a form of self-adaptation for F. In more recent proposals for self-adaptation in DE, some authors have adopted other random distributions, besides the uniform one, such as Gaussian and Cauchy distributions [25].

IV. Experiments

The generation of test data following the constraints-handling formulation can be arbitrarily complex (Section II-B). Nonetheless, this problem can be especially challenging since all the constraints are equalities that cannot be relaxed (otherwise we would not cover the objective branch).

In practice, DE has been found to be very robust, particularly in constrained optimisation, obtaining good results in a wide variety of problems, and requiring a relatively low number of function evaluations. For example, it is worth mentioning that during the special session on constrained optimisation held at CEC 2006 [27], the best approach was based on DE [14], as were three of the top five approaches. Also, there are reported works which focus on keeping low the number of evaluations, with approaches based on DE [11], [12], [13]. With the aim of evaluating the performance of DE in the present problem, we applied different models over a set of benchmark programmes.

We also faced the DE based approach against others. On the one hand, in order to compare DE with other well-known evolutionary algorithms under equal conditions, we ran experiments for the Breeder Genetic Algorithm (BGA) [15]. The BGA has been theoretically studied in a number of works [15], [28], so there are available some guidelines for parameter tuning, and it has been successfully applied to a wide range of real world problems, with different domains [16], [17]. So, we considered BGA as a well-established technique, and therefore suitable for comparisons as a reference approach. On the other hand, we compared the performance of the DE approach with results of other test data generators in the literature over the same programmes.

A. Experimental Setup

All the DE models we have evaluated, as well as the BGA, were embedded in a test data generator following the selection step described in Section II-B.1.

For the evaluation of a solution, the constraint handling mechanism adopted here is that of superiority of feasible points [29]. That is, a feasible solution will always be preferable than an infeasible one. If both solutions are infeasible, that with a lower violation of constraints wins. Finally, if both solutions are feasible, the one with better value in the objective function will win. This is a simple and efficient mechanism in which other works have been based [11], [13].

Independently of the stopping criterion of each algorithm, every execution was halted as soon as a 100000 evaluations were detected. Each experiment was repeated 50 times.

1) Benchmark Programmes: All the programmes we have selected but one are typical case studies used in some other work in the literature [6], [5], [18], [4], [19], [7]; namely, we had access to the source codes of `triangle1`, `atof` and `remainder`, which take integer parameters, and `triangle2`, which is the continuous parameters version of `triangle1`. In order to expand the comparison with real-valued domains, we added `sncndn` to the benchmark, which is a numerical calculus function extracted from [30].

Table I presents some of the programmes' characteristics; these are the number of parameters composing an input, the type of the parameters, the bounds for the values a parameter can take, the number of branches in the source code and the maximum number of critical conditions (constraints) for a branch. In the case of `atof` we codify each character as an integer following the ASCII encoding.

TABLE I
CHARACTERISTICS OF THE PROGRAMMES USED IN THE EXPERIMENTS.

programme	parameters	type	bounds	branches	critical
`triangle1`	3	integer	$[-12500, 12500]$	26	6
`atof`	10	character	$[0, 127]$	30	8
`remainder`	2	integer	$[-33000, 33000]$	18	4
`triangle2`	3	real	$[-12500, 12500]$	26	6
`sncndn`	2	real	$[-500, 500]$	16	2

2) DE Models: With the aim of comparing the performance of typical DE variants, we adopted DE/$rand$/1/bin and DE/$best$/1/bin, with both a fixed value of F, and dithering.

The superiority of feasible points mechanism [29] is applied directly in the binary comparisons of the selection operator.

The performed experiments involved variations in the following parameters: G, the maximum number of generations that a single optimisation can perform; M, the population size; and F, the differential constant for mutation. The values for each parameter were: $G = 100, 200$, because (according to our tests) for some not-so-difficult problems 100 generations are enough, but others required more than that to reach optimal values; $M = 10, 20, 30$, because DE requires smaller populations in order to obtain competitive results compared with other techniques (this is consistent with our results); and $F = 0.25, 0.5, 0.75$, to allow DE perform from a fine coarse search with a small F, to a standard search with an F closer to 1.. For the dithering mechanism, we adopted a Gaussian distribution, with mean of 0.5 and standard deviation of 0.5, as adopted in [25].

Also, in order to discern significant differences among the results, the Kruskal Wallis test was used together with Dunn's post-hoc test where appropriate ($\alpha = 0.05$). These tests allow us to know it the results from different settings may come from the same random distribution. If not, is possible to suggest if a configuration is better than the other; for doing this, we compared mean values.

3) BGA: The BGA can be seen as a combination of Genetic Algorithms and Evolution Strategies [15]. Parent individuals are chosen with truncation selection; in our experiments, we set the truncation threshold to 25%, which is a typical value. Discrete crossover is employed to create the offspring individuals; the probability of crossover is always fixed to 1. The mutation operator was originally designed for continuous variables. A variable x is selected with probability $1/n$, n denoting the number of problem variables. Mutation of at least one variable is forced. x is then added a value in the interval $[-R \cdot (u - l), R \cdot (u - l)]$, with $[l, u]$ denoting the bounds for x. A usual value for R is 0.1 [15], [17]. In our case, however, we may have programmes with integer variables. In order to identify the best parameter value for these programmes we ran preliminary experiments with $R = 0.001, 0.01, 0.1, 0.2$. The best values were obtained with $R = 0.001$ for `triangle1` and `remainder`, and with $R = 0.1$ for `atof`. The implemented stopping criterion is to achieve the optimum or a maximum number of generations. In the experiments, we set the maximum number of generations (G) to 50, 100 and 200.

The superiority of feasible points mechanism [29] is employed to rank the population before selection.

In order to check the performance of the BGA with small and relatively big populations, we made experiments with population sizes $M = 10, 25, 50, 100, 200, 300, 400$.

B. Evaluation of DE models

In Table II are shown the results for the different experiments on the DE approach. More precisely, the tables present the means for the percentage of branches covered (%) and the number of inputs generated (#) during the whole process. Taking priority over the coverage percentage, the best values for each programme are marked in bold. We have only included tables with $G = 100$, for considering representative. However, all the comments in this analysis also hold for the results not included.[1] From the next discussions it will become evident that there are two classes of problems in the selected benchmark, say the *easy* ones and the *hard* ones. We called easy problems to `triangle1`, `triangle2` and `remainder`, for which we have found a combination of parameters which cover 100 % of the branches. On the other hand, we called hard ones to `atof` and `sncndn`, which consume a high number of evaluations to obtain

[1]A document which contains results for all the experiments performed is available at `http://sites.google.com/site/ricardolandabecerra/papers/DEinST-extended.pdf` Such document is available for the interested reader, for it supports all the comments here, while not being mandatory for the comparisons.

a good solution. This simple classification is adopted for better understanding the performance of the algorithm with different parameters.

Let us begin with the comparison of results on the maximum number of generations G. For coverage, only in `triangle2` we found significant differences among different configurations, and also in `sncndn` with DE/$rand$, so DE is robust in G for coverage. Conversely, when analysing the number of inputs, most of the configurations have differences (except in `triangle1` with DE/$best$ and in `reminder` with both), and means suggest that lower values of G are better. So, as a suggestion for tuning, G can be adjusted with a low value, and then increasing it a satisfactory coverage is not achieved. If not enough resources for experimentation are available, a larger value can obtain better results from the beginning.

Now, regarding the population size M, most of the configurations presented significant differences, in both coverage (except for some specific cases in `triangle2` and `remainder`) and inputs (except for some specific cases in `triangle1` and `triangle2`). Furthermore, regarding coverage, means suggested that larger values of M are significantly better, but regarding the number of inputs, the results are mixed (in DE/$rand$, a larger M is suggested to better except in `triangle1`, which is not clear, and in DE/$best$, a larger M is suggested to be better in `triangle1` and `triangle2`, an intermediate value is suggested to be better in `remainder`, and a lower value is suggested to be better in `atof` and `sncndn`). That is to say, in easy problems the effect seems to be uniform, and larger values are commonly better; but in hard problems, the results suggest that a small M can reduce the number of inputs, but also affects the coverage. In general, and giving preference to coverage, a tuning suggestion is to keep larger values of M (around 30).

Proceeding with a specific parameter of DE, let us examine different values of F. First, concerning coverage, $F = 0.25$ exhibited significant differences against the other three values (except in `sncndn`, where the results of $F = 0.25$ may come of the same distribution as those of $F = 0.75$), and the test suggested that such value is significantly worse; while $F = 0.75$ is always among those which are suggested to be the best on easy problems. And concerning the number of inputs, $F = 0.25$ again presented differences against the others (except in `sncndn`, where $F = 0.25$ is similar to 0.75, and in `remainder`, where $F = 0.25$ is similar to 0.5, all for DE/$rand$; and in `triangle1`, `triangle2` and `atof`, for DE/$best$), and $F = 0.75$ also presented differences (except with DE/$best$ in `triangle1`, `triangle2` and `atof`). This time, a larger value is suggested to be better in `triangle1`, `triangle2` and `atof`, while remaining unclear in the rest of the programmes. An overall evaluation returned that $F = 0.25$ is suggested to be significantly worse for the easy programmes, while $F = 0.75$ is among the best values for most of the problems. From those results, we suggest to set a large F, close to 0.75. Interestingly, using dithering presents much variable results, depending heavily on the programme.

In order to ascertain the significantly best parameter configuration for a DE variant, we prioritise over the coverage. That is, we first test for statistically significant differences between every parameter combination and the configuration with highest coverage. Then, we repeat the process with regard to number of inputs, just among those for which no dissimilarity was found. We deem the resulting set of non-different combinations as the best. Once obtained such configurations, we performed a comparison between the DE/$rand$ and DE/$best$ variants. Regarding coverage, there are no significant differences in `triangle1`, `triangle2` and `remainder`, that is, the easy programmes. On the hard ones, there are differences, and the mean coverage is suggested to be better for the DE/$rand$ variant. Now, regarding the number of inputs, only in `triangle1` there are not significant differences. For the rest of the cases, DE/$rand$ is suggested to be better for hard problems, while DE/$best$ is suggested to be better for the easy problems. In general, if some information about the programmes is available, and giving preference to coverage, DE/$best$ variant was found to be more appropriate for the easy ones; the converse can also be mentioned. This is consistent with the fact that the DE/$best$ variants are more greedy, and they focus on exploitation, so if the problem is, say, unimodal and separable (characteristics considered easier than the opposite), an algorithm with good amount of exploitation is convenient; and a more random approach is frequently better for problems with difficult characteristics, as multimodality [31].

C. Comparison with other approaches

1) DE vs. BGA: In order to present a comparison with a reference approach, we present results of BGA in Table III.

Regarding the maximum number of generations, BGA presented significant differences only in some particular cases of `triangle1`, `triangle2` and `remainder` when watching the coverage; and presented differences in most of the cases (except some particular cases of `triangle1`) when watching number of inputs. So, BGA is affected in a similar way than DE, that is to say, a larger value is better.

For M, and regarding coverage, most of the programmes had differences with small population sizes; furthermore, there are not significant differences for $M = 200$, 300 and 400. Regarding number of inputs, results are not clear and is harder to extract patterns. In general, and giving preference to coverage, the means suggest that larger values are better, but as mentioned before, starting from 200 there are not differences.

Now, let us focus on the best results obtained for each technique (obtained as mentioned in the previous section). On coverage, there are no significant differences in `triangle1` for BGA and DE/$rand$, and in the rest of the cases BGA is suggested to be better only on `atof`. And regarding number of inputs, there are no differences in `sncndn` for BGA and DE/$rand$, while the rest presented differences and BGA is

TABLE II

Results for DE/$rand$/1/bin (top) and DE/$best$/1/bin (bottom); $G = 100$.

		`triangle1`		`triangle2`		`atof`		`remainder`		`sncndn`	
		%	#	%	#	%	#	%	#	%	#
$M = 10$	$F = 0.25$	83.54	12616	80.69	18628	57.67	22302	97	800	82.88	5444
	$F = 0.5$	92.77	5842	92	6481	72.47	15119	99.44	301	85.38	4654
	$F = 0.75$	99.85	619	97	1885	90.13	7284	99.89	273	82.5	6232
	$dithering$	99.46	696	96.08	1841	69.8	50740	99.89	373	81.63	6555
$M = 20$	$F = 0.25$	92.77	5825	93.15	13464	76.93	25958	99.89	283	84.75	10605
	$F = 0.5$	**100**	**491**	**99.69**	**2141**	93.27	8212	100	326	88.75	7000
	$F = 0.75$	100	794	98.54	3316	98	4842	100	450	84.5	10606
	$dithering$	100	923	96.23	3708	88.67	17279	100	486	81.88	13312
$M = 30$	$F = 0.25$	97.92	2500	95.46	8443	85.27	22097	**100**	**323**	86.88	13224
	$F = 0.5$	100	655	99.92	2991	98.27	5365	100	455	**89.25**	**10151**
	$F = 0.75$	100	1113	98.85	5094	**99.6**	**5798**	100	622	87.25	12394
	$dithering$	100	1325	96.23	5698	95.8	10075	100	687	81.75	20210
$M = 10$	$F = 0.25$	78.92	16121	77.69	23972	45.4	29121	96	908	81.38	5282
	$F = 0.5$	96.15	3279	89.77	8922	52.93	27586	99.78	231	81.75	5869
	$F = 0.75$	99.54	643	96.85	2048	63.93	23289	99.33	373	81.75	5949
	$dithering$	95.54	5557	93	13116	47.73	91863	99.33	292	81.5	5929
$M = 20$	$F = 0.25$	94.15	3846	93.39	12636	51.2	53100	**100**	**233**	82.38	10847
	$F = 0.5$	99.77	619	99.69	4093	63	40088	100	331	83.63	10578
	$F = 0.75$	100	804	99	3270	70.73	28557	100	439	82.5	12665
	$dithering$	99.92	518	99.77	1982	56.93	83054	100	292	81.5	11978
$M = 30$	$F = 0.25$	96.77	3457	95.62	9842	57.93	64540	100	314	83.88	14756
	$F = 0.5$	100	662	**100**	**2937**	64.93	54519	100	460	**86**	**13213**
	$F = 0.75$	100	1062	98.23	5039	**82.27**	**29296**	100	597	83.88	16772
	$dithering$	**100**	**587**	100	2558	60.93	84212	100	402	81.75	18210

worse, (except in the hard problems, where is among the bests). In general, and giving preference to coverage, BGA was only better than DE in `atof`, a hard problem.

2) DE approach vs. others: Other SBSTDG works in the literature have used some of the benchmark programmes herein to evaluate their approaches for branch coverage. The search algorithms adopted include GAs [5], [6], Scatter Search [18], [19] and Estimation of Distribution Algorithms [4]. All the test data generators in these works conform to the classical approach (Section II-A). In [7], preliminary experiments were conducted using a GA in nine different test data generators: two based on the classical approach, four based on constrained optimisation (four penalty functions) and three based on multiobjective optimisation (transforming constraints into objectives). Table IV shows the best outcomes of these works and DE for our programmes. Best results are marked in bold.

As it can be observed, the DE generator clearly obtains the best values among the constraints based approaches in all programmes but `atof`. In this case, DE is the second best, close to the value in [4], which corresponds to a GA with static penalty function. Compared with classical approaches, the DE generator notably beats all but the Scatter Search approach from [19] in programme `remainder`, where DE is the second best. All in all, the DE approach presents the best or second best results in every programme.

V. Conclusions

In this work, we have applied DE to the problem of automatically generating a set of test inputs that cover all the branches in the source code of a programme. Building upon a novel constraint-handling formulation, we have evaluated empirically the performance of a number of popular DE models. Also, we have faced DE with a well-known Genetic Algorithm, i.e. BGA, and other test data generators in the literature.

The results obtained by DE are better in most of the problems than those obtained by BGA, but properly setting the parameters is an important task. DE appears specially sensitive to the value of F. Parameters as G and M only need to be large enough to allow DE perform the search accordingly with the difficulty of the problem. The variant DE/$best$/1/bin is recommended if the problem is not so difficult, but in case of lack of information, DE/$rand$/1/bin can also work well in a wide variety of problems.

Combining advantages of different variants, is a possible future work which may improve the performance of DE in this field. The use of mechanisms which adaptively provide this combinations, by incorporating information extracted from the problem, are specially suitable. So, a cultural algorithm framework is a good candidate for future research.

Acknowledgement

The first author acknowledges support from CONACyT through a postdoctoral fellowship for research at the University of Birmingham, UK. This work is also supported by an EPSRC grant (EP/D052785/1).

References

[1] B. Beizer, *Software Testing Techniques*. New York: Van Nostrand Rheinhold, 1990.

[2] P. McMinn, "Search-based software test data generation: a survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[3] G. McGraw, C. Michael, and M. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, 2001.

[4] R. Sagarna and J. A. Lozano, "On the performance of estimation of distribution algorithms applied to software testing," *Applied Artificial Intelligence*, vol. 19, no. 5, pp. 457–489, 2005.

TABLE III

RESULTS FOR BGA; $G = 100$.

	`triangle1`		`triangle2`		`atof`		`remainder`		`sncndn`	
	%	#	%	#	%	#	%	#	%	#
$M = 10$	92.46	8630	92.31	2832	70.47	17436	95.22	2180	81.25	4970
$M = 25$	97.54	6214	92.31	6674	76	31139	99.56	1760	81.25	12525
$M = 50$	99.8	3635	92.39	15093	86.8	33410	100	1794	81.38	25162
$M = 100$	**100**	**3208**	92.31	28104	91.8	28876	**100**	**1658**	81.38	49920
$M = 200$	100	4200	**92.54**	**57524**	99.2	11428	100	2484	81.38	99812
$M = 300$	100	5064	92.39	73734	99.13	10614	100	2988	**81.5**	**100002**
$M = 400$	100	6160	92.39	89016	**99.87**	**8000**	100	3448	81.25	100400

TABLE IV

BEST RESULTS BY DIFFERENT TEST DATA GENERATORS. $pen^{[7]}$ AND $mo^{[7]}$ REFER TO THE PENALTY AND MULTIOBJECTIVE APPROACHES IN [7],

		classical approach						constraints approach		
programme		[6]	[5]	[18]	[4]	[19]	[7]	$pen^{[7]}$	$mo^{[7]}$	DE
`triangle1`	%	100	-	96.15	100	100	94.36	95.39	93.46	**100**
	#	16915	-	2698	6150	1108	22280	19835	15917	**468**
`triangle2`	%	100	-	96.15	100	100	-	-	-	**100**
	#	42086	-	2170	6200	4250	-	-	-	**1859**
`atof`	%	100	-	100	100	91.33	100	**100**	98.56	100
	#	35263	-	13509	7685	570306	5920	**5040**	13915	5309
`remainder`	%	-	100	100	100	**100**	89.44	89.63	89.07	100
	#	-	944	482	2360	**141**	18870	13527	18870	233

[5] H. Sthamer, "The automatic generation of software test data using genetic algorithms," Ph.D. dissertation, University of Glamorgan, Pontypridd, Wales, Great Britain, 1996.

[6] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[7] R. Sagarna and X. Yao, "Handling constraints for search based software test data generation," in *Proceedings of IEEE International Workshop on Search Based Software Testing*, 2008, pp. 232–240.

[8] K. Price, "An introduction to differential evolution," in *New Ideas in Optimization*, D. Corne, M. Dorigo, and F. Glover, Eds. London, UK: McGraw-Hill, 1999, pp. 79–108.

[9] R. Storn, "Differential evolution research - trends and open questions," in *Advances in Differential Evolution*, ser. Studies in Computational Intelligence, U. K. Chakraborty, Ed. Springer, 2008, pp. 1–31.

[10] J. Lampinen and I. Zelinka, "Mechanical engineering design optimization by differential evolution," in *New ideas in optimization*. Maidenhead, UK: McGraw-Hill Ltd., UK, 1999, pp. 127–146.

[11] J. Lampinen, "A Constraint Handling Approach for the Diifferential Evolution Algorithm," in *Proceedings of the Congress on Evolutionary Computation 2002 (CEC'2002)*, vol. 2. IEEE Service Center, May 2002, pp. 1468–1473.

[12] R. Landa Becerra and C. A. Coello Coello, "Optimization with Constraints using a Cultured Differential Evolution Approach," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2005)*, vol. 1, Washington DC, USA. New York: ACM Press, June 2005, pp. 27–34.

[13] E. Mezura-Montes, C. A. Coello Coello, and E. I. Tun-Morales, "Simple Feasibility Rules and Differential Evolution for Constrained Optimization," in *Proceedings of the 3rd Mexican International Conference on Artificial Intelligence (MICAI'2004)*. Springer Verlag, April 2004, pp. 707–716, lNAI 2972.

[14] T. Takahama and S. Sakai, "Constrained optimization by the e constrained differential evolution with gradient-based mutation and feasible elites," in *IEEE Congress on Evolutionary Computation, 2006. CEC 2006*, 2006, pp. 1–8.

[15] H. Mühlenbein and D. Schlierkamp-Voosen, "Predictive models for the breeder genetic algorithm i. contiunous parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 25–49, 1993.

[16] U. Bartling and H. Mühlenbein, "Optimization of large scale parcel distribution systems by the breeder genetic algorithm," in *Proceedings of the 7th International Conference on Genetic Algorithms*, T. Bäck, Ed. Morgan Kaufmann, 1997, pp. 473–480.

[17] R. S. O. Montiel, O. Castillo and P. Melin, "Application of a breeder genetic algorithm for finite impulse filter optimization," *Information Sciences*, vol. 161, no. 3–4, pp. 139–158, 2004.

[18] R. Blanco, J. Tuya, and B. Adenso-Díaz, "Automated test data generation using a scatter search approach," *Information and Software Technology*, vol. 51, pp. 708–720, 2009.

[19] R. Sagarna and J. A. Lozano, "Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms," *European Journal of Operational Research*, vol. 169, no. 2, pp. 392–412, 2006.

[20] N. E. Fenton, "The structural complexity of flowgraphs," in *Graph Theory with Applications to Algorithms and Computer Science*, Y. Alavy, G. Chartrand, L. Lesniak, D. R. Lick, and C. E. Wall, Eds. New York: John Wiley & Sons, 1985, pp. 273–282.

[21] M. Schoenauer and S. Xanthakis, "Constrained ga optimization," in *Proceedings of the 5th International Conference on Genetic Algorithms*, S. Forrest, Ed. San Mateo, CA: Morgan Kauffmann, 1993, pp. 573–580.

[22] P. McMinn, D. Binkley, and M. Harman, "Empirical evaluation of a nesting testability transformation for evolutionary testing," *ACM Transactions on Software Engineering Methodology*, to appear.

[23] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, pp. 99–118, 1977.

[24] U. K. Chakraborty, *Advances in Differential Evolution*, ser. Studies in Computational Intelligence. Springer, 2008.

[25] Z. Yang, K. Tang, and X. Yao, "Self-adaptive differential evolution with neighborhood search," in *IEEE Congress on Evolutionary Computation, CEC 2008. (IEEE World Congress on Computational Intelligence)*, June 2008, pp. 1110–1116.

[26] S. Das, A. Konar, and U. K. Chakraborty, "Two improved differential evolution schemes for faster global search," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2005*, 2005, pp. 991–998.

[27] J. J. Liang, T. P. Runarsson, E. Mezura-Montes, M. Clerc, P. N. Suganthan, C. A. C. Coello, and K. Deb, "Problem definitions and evaluation criteria for the cec 2006 special session on constrained real-parameter optimization," Nanyang Technological University, Singapore, Tech. Rep., 2006.

[28] H. Mühlenbein, "The equation for response to selection and its use for prediction," *Evolutionary Computation*, vol. 5, no. 3, pp. 303–346, 1997.

[29] K. Deb, "An Efficient Constraint Handling Method for Genetic Algorithms," *Computer Methods in Applied Mechanics and Engineering*, vol. 186, no. 2/4, pp. 311–338, 2000.

[30] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C. The Art of Scientific Computing*. New York: Cambridge University Press, 1988.

[31] V. Feoktistov, *Differential Evolution: In Search of Solutions*, ser. Springer Optimization and Its Applications. Secaucus, NJ, USA: Springer, 2006.