Orthogonal Exploration of the Search Space in Evolutionary Test Case Generation

Fitsum M. Kifetew
Fondazione Bruno Kessler
Trento, Italy
kifetew@fbk.eu

Andrea De Lucia
University of Salerno
Fisciano (SA), Italy
adelucia@unisa.it

Rocco Oliveto
University of Molise
Pesche (IS), Italy
rocco.oliveto@unimol.it

Annibale Panichella
University of Salerno
Fisciano (SA), Italy
panichella@unisa.it

Paolo Tonella
Fondazione Bruno Kessler
Trento, Italy
tonella@fbk.eu

1. INTRODUCTION

Search based techniques have been successfully applied to several areas of Software Engineering in general, and Software Testing in particular [14]. Specifically, a class of algorithms known as Evolutionary Algorithms (EAs) has been widely used in automating the generation of test data [21].

In evolutionary test case generation [27], candidate test cases are encoded into a population of individuals (solutions). These individuals are then evaluated by executing them against the System Under Test (SUT) and their fitness (goodness) is measured with respect to a given criterion, e.g., branch coverage. Evolutionary operators such as selection, crossover and mutation are then used to evolve a new generation of individuals with better qualities than their parents. The process is repeated for several generations either until the criterion is satisfied or the fixed search budget is finished. A particularly popular flavor of EA are Genetic Algorithms (GAs) [11], quite often and successfully adopted in the testing community [2, 10, 26, 27]. The success of evolutionary test case generation techniques in general, and GAs in particular, depends on several factors. One of these factors is the level of diversity among the individuals in the population, which directly affects the exploration ability of the search. Indeed, given the fact that the search space is usually extremely large, the GA has to maintain an adequate level of diversity in the population in order to effectively explore the search space and look for alternative, potential solutions. The basic genetic operators play an important role in maintaining this diversity [13]: crossover is generally considered as a background operator to maintain diversity for exploration, while selection and mutation are used to exploit the current solution to find nearby better ones [24]. However, these operators might not suffice by themselves in maintaining enough diversity. Depending on the difficulty of the current search target and the type of selection scheme in use, individuals in the population can become too similar to one another, eventually converging to a single, sub-optimal individual. Hence, the ability of the search to explore new areas of the search space is greatly reduced. This phenomenon is referred to as genetic drift [23].

Evolutionary test case generation techniques that employ GAs could be severely affected by genetic drift. Especially for hard-to-cover program branches, it can happen that nearly all the candidate individuals in the search become similar. Consequently, genetic operators simply recombine genetic material among these similar individuals, resulting in convergence to a sub-optimal solution, which does not achieve the desired objective. Hence, test data generation

ABSTRACT

The effectiveness of evolutionary test case generation based on Genetic Algorithms (GAs) can be seriously impacted by genetic drift, a phenomenon that inhibits the ability of such algorithms to effectively diversify the search and look for alternative potential solutions. In such cases, the search becomes dominated by a small set of similar individuals that lead GAs to converge to a sub-optimal solution and to stagnate, without reaching the desired objective. This problem is particularly common for hard-to-cover program branches, associated with an extremely large solution space.

In this paper, we propose an approach to solve this problem by integrating a mechanism for orthogonal exploration of the search space into standard GA. The diversity in the population is enriched by adding individuals in orthogonal directions, hence providing a more effective exploration of the solution space. To the best of our knowledge, no prior work has addressed explicitly the issue of evolution direction based diversification in the context of evolutionary testing. Results achieved on 17 Java classes indicate that the proposed enhancements make GA much more effective and efficient in automating the testing process. In particular, effectiveness (coverage) was significantly improved in 47% of the subjects and efficiency (search budget consumed) was improved in 85% of the subjects on which effectiveness remains the same.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Search based testing; test case generation; orthogonal exploration; genetic algorithms; genetic drift
techniques need to properly address this problem if they are to succeed on programs with complex logic and branches.

Previous work treated the problems of diversity considering simple heuristics promoting diversity between individuals within the same generation [1, 8]. Hence, new offsprings might explore regions that have been explored in previous generations while new potential regions can still remain unexplored, increasing the likelihood to reach sub-optimal solutions. This shortcoming motivates our work and this paper is an attempt towards a solution for this problem. Specifically, we integrate in the standard GA an orthogonal exploration mechanism of the search space. Such a mechanism—through the estimation of the evolution directions via Singular Value Decomposition (SVD) [19]—is able to augment the population diversity.

Results achieved on 17 Java classes extracted from well-known libraries show that with the appropriate application of diversification techniques, the effectiveness and efficiency of GAs in structural test data generation can be greatly improved. In particular, effectiveness (coverage) was significantly improved in 47% of the subjects and efficiency (search budget consumed) was improved in 85% of the subjects on which effectiveness remains the same.

The key contributions of this paper can be summarized as follows:

1. For the first time, we apply an SVD-based GA to the problem of test case generation, addressing the problem of stagnation during the search for test input data.
2. We propose three variants of SVD-based GA, to overcome a specific problem occurring in SVD-based GA applied to test data generation (namely, the evolution direction becoming null).
3. We make the implementation of our approach available as an extension of the search based testing tool EvoSuite.
4. We evaluate our approach on a set of publicly available programs.

The remainder of this paper is organized as follows. Section 2 discusses relevant related works. Sections 3 and 4 present our proposed diversification techniques and their implementation, respectively. Sections 5 and 6 present the design and the results achieved in our empirical study, while the discussion of the threats that could affect the results achieved is reported in Section 7. Finally, Section 8 points out future works and concludes the paper.

2. RELATED WORK

This section presents a survey of the related works about (i) evolutionary test data generation and (ii) diversification for meta-heuristic approaches in order to improve their optimality and convergence speed.

2.1 Evolutionary Test Case Generation

The application of search based techniques for automated test case generation has received considerable attention in recent years from the testing community. As a result, a large body of research works has been reported where different meta-heuristic techniques have been applied to address the problem of test data generation [21]. Evolutionary techniques in particular, and specifically GA, have been intensively investigated for the purpose of structural testing [2, 10, 26, 27].

Various aspects of evolutionary test data generation, such as selection methods, crossover and mutation operators, fitness evaluation schemes, etc., have been studied in the literature [21]. The problem of genetic drift has been discussed in the literature as an issue related with GAs in general [19, 23]. There have also been research efforts aimed at increasing diversity among test cases following varying criteria for measuring similarity between test cases. One such approach [1] relies on sampling the program data-state, by tracing variables via intensive instrumentation, and looking for test cases that result in a different state. Another approach [8] for measuring diversity among test cases was proposed based on the notion of Information Distance. However, in both works, the approaches measure how different an individual test case is from the other test cases within the current population, hence, test cases belonging to previously explored regions can still be generated. In contrast, our approach introduces diversity by exploring areas of the search space which are different from those traversed so far, based on the direction through which the population as a whole has evolved.

2.2 Search Diversification in Meta-heuristics

In recent years considerable effort has been devoted to improving the convergence rate and solution accuracy of meta-heuristic algorithms. Diversity preserving techniques have been demonstrated to be particularly useful for increasing diversity between individuals within populations of solutions, hence improving the performance of GA [23]. One simple technique for introducing diversity consists of modifying the mutation rate. Higher mutation rate clearly increases the ability to explore more areas, however, it prevents the convergence toward any optimum solution, reducing the ability to locally searching for nearby better solutions [24]. Moreover, this methodology might not guarantee the exploration of new unexplored regions and several regions can be revisited many times. Hence, several enhancing techniques have been proposed to promote diversity by (i) using novel and specialized genetic operators [7, 12, 15] or (ii) introducing new genetic materials [3, 4].

Niching methods, based on the mechanism of natural ecosystems, are the first diversity preserving techniques proposed in literature when dealing with traditional numerical optimization problems [15]. Among the different niching methods, one of the most known and used methods is fitness sharing [15] which modifies the fitness function by making worse the fitness values in densely populated regions [12]. Another method for dealing with population drift is the distance crowding schema [7] which replaces individuals which are quite similar to each other by inserting new individuals, and in its improved variant by introducing competition between children and parents of identical niches [20]. All these techniques work by introducing diversity using just information about each generation separately and ignoring any information about the evolution history of the search process. Thus, even if they introduce diversity, the problem of generating individuals (solutions) already considered in the past still remains.

Recently, De Lucia et al. [19] proposed to use the information about the history of evolution for generating new individuals which explore new regions that were not explored in the past few generations. Indeed, the movements of individuals across different generations—called evolution directions—are caught using SVD [25], which gives an estimate of where the search is going to. Thus, diversity is introduced by replacing the worst individuals (individuals with the lowest fitness values) with new individuals from orthogonal evolution directions which will explore new search regions.

In order to address the problem of loss of diversity, Battiti et al. [3] proposed to check for the repetition of solutions and added this scheme to the Tabu Search algorithm. With the new Tabu scheme, called Reactive Tabu Search, an appropriate size of the tabu list is learned in an automated way by reacting to the occurrence of
cycles. If a diversity loss is diagnosed, i.e., several individuals are similar (or identical) to each other, then the search is diversified by making a number of random moves proportional to the moving average of the cycle length.

In this work, we propose a novel GA customized for test data generation that combines both SVD-based GA and reactive exploration (loosely inspired by Reactive Tabu Search) for improving both effectiveness and efficiency of evolutionary testing. We focused on SVD-based GA and reactive exploration because they do not modify the fitness function, differently from the niching mechanisms, which are quite hard to apply to evolutionary test case generation. Moreover, we preferred SVD-based GA to the distance crowding schema because the SVD-based GA has been demonstrated to be significantly better than the distance-crowding based GA [19].

To the best of our knowledge, the approach proposed in this paper is the first attempt to address the genetic drift problem in evolutionary testing.

3. DIVERSIFYING THE EXPLORATION

Maintaining an adequate level of diversity in the search is a key factor for the effectiveness of a GA. Indeed, a scarcely diversified population can make the genetic operators (crossover and mutation) unable to produce offspring outperforming their parents, with a tendency of guiding the search toward some local optimum. In the presence of limited diversity in the population, GA tends to generate offspring quite similar to their parents, creating a limited number of groups of solutions (niches) that are close to each other within the search space, while other parts of the search space are left unexplored. This phenomenon, called premature convergence or genetic drift, is a well known problem when solving optimization problems with GA [28].

Taking into account the randomness of search space exploration introduced by the mutation and crossover operators, the use of SVD has been proposed to estimate the directions along which the best individuals in the population are evolving across generations [19]. Such movements of individuals in the search space are called evolution directions. Once estimated, these directions are used for (i) accelerating the evolution towards promising directions (i.e., directions currently followed by the individuals having the best fitness values); and, at the same time, (ii) increasing the exploration of new regions in the search space, by introducing new individuals with orthogonal evolution directions [19].

The idea is that a population of solutions \( P \) obtained by a GA at a given generation \( t \) can be viewed as an \( m \times n \) matrix:

\[
P_t = \begin{bmatrix}
p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\
p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\
& \vdots & \ddots & \vdots \\
p_{m,1} & p_{m,2} & \cdots & p_{m,n}
\end{bmatrix}
\]  

where \( n \) is the number of genes of each individual, \( m \) is the population size, while the generic entry \( p_{i,j} \) denotes the value of the gene \( x_j \) for the \( i \)th individual. SVD can be used to capture how the population is distributed within the search space: (i) the axes along which the population exhibits the most variations (eigenvectors) and (ii) the magnitude or importance of such axes (eigenvalues). Since two populations are differently distributed within the search space, they exhibit different SVD decompositions. Hence, a way to estimate how the population is evolving during \( k \) generations consists of comparing the corresponding SVD decompositions before and after \( k \) generations. Formally, let \( P_t \) and \( P_{t+k} \) be two different populations generated by GA at generations \( t \) and \( t + k \), respectively (where \( k \) is a user defined parameter). SVD is computed on both populations, resulting in matrices that represent evolution directions and magnitudes: \( P_t = (U_t \cdot \Sigma_t \cdot V_t^T) \) and \( P_{t+k} = (U_{t+k} \cdot \Sigma_{t+k} \cdot V_{t+k}^T) \) where \( V \in \mathbb{R}^{n \times n} \) and \( \Sigma \in \mathbb{R}^{n \times n} \). The column vectors of \( V \) represent the main directions in which the population is distributed while the diagonal matrix \( \Sigma \) represents the importance of each direction in \( V \). Computing \( \Sigma_{t+k} - \Sigma \) gives a precise indication of the direction of evolution of the population, while \( \Sigma = \Sigma_{t+k} - \Sigma \) indicates its magnitude. New individuals in orthogonal spaces are generated as follows:

\[
P'_{t+k} = U_{t+k} \cdot (\Sigma_{t+k} + \Sigma) \cdot (V_{t+k} + \text{Orth}(V))^T
\]

where Orth(\( V \)) is a matrix whose column vectors are unit and orthogonal with respect to the column vectors of \( V \). There is no unique way to generate orthogonal column vectors. We have chosen the simple method shown in Algorithm 1. Such an algorithm creates orthogonal vectors in a simple manner: given a vector \( \vec{v} \), its orthogonal vector can be computed by (i) creating a new vector \( \vec{v}' \) with the same elements in \( \vec{v} \) but in reverse order and (ii) multiplying by -1 the 50% of its elements. Once the new orthogonal individuals are generated, the worst individuals in \( P_{t+k} \) (i.e., in the last generation) are replaced by these new individuals, introducing diversification in the population along orthogonal, and potentially unexplored, directions.

\begin{algorithm}
\begin{algorithmic}
\State \text{sign} \leftarrow 0;
\State \text{odd} \leftarrow \text{true};
\State \text{morth} \leftarrow \text{createMatrix}(\text{numRows}, \text{numCols});
\If{\text{mod}(\text{numRows},2) = 0}
\State \text{odd} \leftarrow \text{false};
\EndIf
\For{\text{r} = 0; \text{r} < \text{numRows}; \text{r} \text{ + } \text{+} \Do
\If{\text{c} = 0; \text{c} < \text{numCols}; \text{c} \text{ + +} \Do
\State \text{sign} \leftarrow \text{(r < numRows/2)}? 1 : 1;
\If{\text{(odd = true) AND (r = numRows/2)}}
\State \text{sign} \leftarrow 0;
\EndIf
\EndIf
\EndFor
\EndFor
\end{algorithmic}
\caption{orth: generates a matrix whose column vectors are orthogonal with respect to those of the matrix \( m \) passed to it as parameter, with numRows rows and numCols columns}
\end{algorithm}

Figure 1: SVD-based GA: Graphical interpretation of orthogonal diversification.
Figure 1 shows the graphical interpretation of the evolution direction followed by a population $P$ between generations $t$ and $t+k$. We can see how $|\Sigma_{t+k} + \Sigma|$ allows to generate individuals which are shifted in the search space, while $(V_{t+k} + \text{Orth}(V))$ creates new individuals with orthogonal evolution directions, i.e., new individuals which explore orthogonal and unexplored regions of the search space.

In this paper, we propose to use a SVD-based GA for evolutionary test data generation. Starting from its basic definition, we extend the algorithm proposing three novel variants, in order to make the SVD-based GA more suitable for the test data generation problem. Specifically, in this work we introduce (1) history aware orthogonal exploration of the search space, (2) reactive exploration and (3) a combination of the two approaches. These extensions are motivated by the fact that the orthogonal exploration of the search space based on SVD alone relies on the existence of a “direction” of evolution across two different generations. Since usually in test data generation there are only a few input vectors which cause the coverage of uncovered branches (especially for hard-to-cover branches) and the fitness landscape around them is often flat, the associated lack of variation in fitness values leaves the search with no evolution directions (or zero directions), resulting in a high likelihood of failure to find orthogonal test data using just the SVD-based scheme.

From a preliminary exploratory investigation of SVD-based GA for structural testing, we have noticed that this case is quite common. Hence, to effectively utilize the potential of SVD-based GA in the context of testing, we need to amend this problem. Our three proposed extensions to overcome this problem are described in the next subsections.

### 3.1 History Aware Orthogonal Exploration

The first extension of SVD-based GA is represented by a history aware approach, in which we consider not only the population at generation $t$ and $t+k$, but also the history of older populations encountered during evolution. Specifically, if $V_{t+k}$ computed between $P_{t+k}$ and $P_t$, gives a zero matrix, we compute $V_t$ between population $P_{t+k}$ and population $P_{t-k}$, $P_{t-2k}$, $P_{t-3k}$, ... until we achieve a non zero direction matrix or we consume the available history entirely. In other words, we propose to compute the evolution directions starting from the last generation $t+k$ going back across the previous generations until we reach a non-null evolution direction (or we exhaust the history). For practical purposes, we consider a bounded history of snapshots: $P_t, P_{t-k}, P_{t-2k}, \ldots, P_{t-hk}$, with $h >= 0$ a small, user defined constant. In our empirical experiments, the value $h = 10$ gives reasonably good results. In the following, this approach is referred to as H-OV-GA.

### 3.2 Reactive Exploration

Another way to ensure diversity in the exploration of the search space consists of taking a random direction, rather than an orthogonal direction, when stagnation —i.e., null evolution direction— occurs. We compute the direction matrix in a similar manner as orthogonal direction, when stagnation —i.e., null evolution direction— occurs. We compute the direction matrix in a similar manner as orthogonal direction, when stagnation —i.e., null evolution direction— occurs. We compute the direction matrix in a similar manner as orthogonal direction, when stagnation —i.e., null evolution direction— occurs. We compute the direction matrix in a similar manner as orthogonal direction, when stagnation —i.e., null evolution direction— occurs. We compute the direction matrix in a similar manner as orthogonal direction, when stagnation —i.e., null evolution direction— occurs.

### 3.3 Reactive Orthogonal Exploration

While the two approaches discussed in the previous sub-sections are expected to improve over the basic algorithm in diversifying the search, their combined effect could be even better than either of them applied separately. Thus, we propose yet another SVD-based GA variant which combines the history aware approach with the reactive exploration. Specifically, we first compute the direction matrix $V_t$ between the current populations $P_{t+k}$ and $P_t$, as previously discussed. If this direction matrix is non-zero, we compute an orthogonal matrix $\text{Orth}(V_t)$ and generate a new population by applying Equation 2. Otherwise, if the direction matrix $V_t$ is equal to zero —i.e. there is no evolution direction— we generate a random direction matrix and generate a new population by applying Equation 3. In the following, this approach is referred to as R-OV-GA.

### 4. IMPLEMENTATION

While the proposed orthogonal exploration approach can be applied to any numeric program, whether it is object oriented or not, for the sake of demonstration we present the implementation of the technique in the context of object oriented programs. Specifically, we assume the goal is to achieve full branch coverage of a Class Under Test (CUT).

In our approach, input vectors contain numeric values and have fixed size. We plan to extend our approach to variable size input vectors and to non-numeric data as part of our future work. To make our approach applicable to an object oriented CUT we make the assumption that any method invocation sequence necessary to prepare the CUT for the execution of the method under test is performed inside the class constructor. If no such constructor exists, we define one, just for the purposes of testing. In this way, a fixed size input vector is required to perform the preparatory method sequence invocation from the constructor. The input parameters used in the preparatory method sequence are added to the signature of such constructor.

Let us assume the constructor of the CUT has $i$ numeric parameters (including those required by the preparatory method sequence invocation). For each target branch $r$ in method $m$ with $j$ numeric parameters, we set up a GA and search for a test case that covers $r$. An individual in the GA is composed of a genotype and a phenotype. The genotype is a vector of dimension $n = i + j$. The phenotype is a sequence of Java statements that are actually executed against the CUT via JUnit. Consequently, the population of $m$ individuals will be an $m \times n$ matrix. We then apply either H-OV-GA, R-GA or R-OV-GA (collectively, these three approaches are indicated as OrthoGA), to find a numeric input vector that covers the target branch $r$.

The main algorithm is shown in Algorithm 2 while the OrthoGA algorithm is shown in Algorithm 3.

#### Algorithm 2 OrthoGA: main algorithm

```java
while total_budget not finished do
    for every target not covered do
        budget ← total_target / uncovered_targets
        solution ← generateSolution(target, budget);
        suite ← suite ∪ solution;
    end for
end while
return suite;
```

where $\text{Rand} \in \mathbb{R}^{n \times n}$ is a randomly generated matrix representing a new random direction. Otherwise, the search continues normally without any random or reactive exploration. In the following, this approach is referred to as R-GA.
Algorithm 3 OrthoGA: generateSolution routine; this routine searches a solution for a given coverage target using budget statement executions

```java
public class A {
    private int a, b, c;
    public A(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public static int m2(double d1, double d2, double[] d3) {
        // Implementation...
    }
    public void m1(int x) {...}
}
```

Figure 2: An example of class under test

Algorithm 2 distributes the available test case generation budget (measured in number of statements executed) across the targets yet to be covered and relies upon generateSolution to obtain an input vector to cover the selected target. Algorithm 3 shows the pseudocode of generateSolution. It includes the common steps performed in evolutionary testing (generation of an initial random population and its evolution by selection, mutation and crossover, depending on the fitness values of the individuals). In addition to such steps, the algorithm periodically (after k generations, with k a user defined parameter) performs orthogonal diversification of the population. Based on a history of length h, an orthogonal population $P_t^*$ is produced and is used to replace the individuals with lowest fitness values. The routine that generates such orthogonal population, named generateOrtho, depends on the specific instance of the OrthoGA algorithm, which can be either H-OV-GA, R-GA or R-OV-GA.

When the method under test m is not a publicly visible method (e.g., it is a private method), we select a public method $m'$ which (directly or indirectly) calls m, if one exists. Otherwise, the target is deemed unreachable. If m is a static method, then the individual will be composed of only the j parameters of m. The actual test case will then be composed of statements for instantiating the class, followed by an invocation of method m on the instance, or simply the static invocation of m if it is a static method.

Let us consider class A in Figure 2 and a target branch inside method m1. An individual in the population could be (3, 4, 5, 0) with the corresponding test case: A a0 = new A (3, 4, 5); A0.m1 (0); For a target inside method m2 an individual could be (10.5, 12.0, 0.5, 1.25) with the corresponding test case:

- double[] double0 = new double[3];
- double0[0] = 12.0;
- double0[1] = 0.5;
- double0[2] = 1.25;

int int0 = A.m2 (10.5, 12.0, double0);

To obtain fixed size input vectors, arrays are instantiated with a fixed, predefined length (3, in this example).

Genetic operations are applied to the vector representation of an individual. In particular, crossover between two individuals is performed by exchanging portions of the vectors representing the genotype. Mutation on an individual changes one or more values from the vector. Each value in the vector is subject to mutation with probability 1/len where len is the length of the vector representing the genotype. The actual mutation operation performed on a value selected for mutation could be one of the following three operations depending on their respective probabilities:

- negate: change the sign of the value;
- increment: a randomly generated delta value is added/subtracted to/from the value;
- replace from pool: replace by a randomly selected value from a pool of previously used values.

Elitism is used, so as to ensure that the elite of each generation is preserved.

Fitness evaluation is based on the well established approach level and branch distance metrics [21] in which the test case associated with an individual is executed against the instrumented CUT. Approach level measures how many control branches away the current individual is from hitting the target branch, while branch distance measures how far the branch condition is from making the branch condition true, where the considered branch is the first one which makes the target no longer reachable. After the execution, the fitness of each individual is computed as the sum of the approach level and the normalized branch distance. This value is minimized, with value zero associated with the target branch being covered by the individual.

Our prototype tool for the branch coverage testing of Java classes is implemented as an extension of the EvoSuite [9] tool and is available for download from www.evosuite.org/orthogonal-ga.

5. EMPIRICAL EVALUATION

This section describes the study we conducted aiming at evaluating whether the performance of evolutionary test case generation improves when the employed GA is enriched with orthogonal and/or reactive exploration mechanisms. The quality focus of the study is the effectiveness and the efficiency of the test case generation process based on the proposed approach compared to standard GA (St-GA hereafter), while the perspective is of researchers aiming at developing more effective automatic testing tools, that could be used by software engineers for reducing the time/cost required for testing a software system.

5.1 Subjects

The experimental study is performed on 17 publicly available Java classes with numerical input data type. Table 5 shows the characteristics of the classes considered in our study in terms of number of methods and number of coverage goals (i.e., branches).

Most of the classes (9) were extracted from the open source Apache.commons.math library, which contains several mathematical and statistical Java classes addressing the most common problems not available in the Java standard libraries or in Commons Lang\(^1\). We also considered 2 classes extracted from the Apache.

\(^1\)http://commons.apache.org/math/
Table 1: Subjects used in our study.

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Origin</th>
<th>Methods</th>
<th>Coverage Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ArithmeticUtils</td>
<td>Apache Commons Math</td>
<td>10</td>
<td>99</td>
</tr>
<tr>
<td>2</td>
<td>Arrays</td>
<td>Java Collections</td>
<td>8</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>Beta</td>
<td>Apache Commons Math</td>
<td>11</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>CreditCardValidator</td>
<td>Apache Commons Valid</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Complex</td>
<td>Apache Commons Math</td>
<td>29</td>
<td>126</td>
</tr>
<tr>
<td>6</td>
<td>FastMath</td>
<td>Apache Commons Math</td>
<td>3</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>Fraction</td>
<td>Apache Commons Math</td>
<td>28</td>
<td>108</td>
</tr>
<tr>
<td>8</td>
<td>IPAddressValidator</td>
<td>Apache Commons Valid</td>
<td>3</td>
<td>243</td>
</tr>
<tr>
<td>9</td>
<td>LDUDecomposition</td>
<td>JAMA library</td>
<td>8</td>
<td>76</td>
</tr>
<tr>
<td>10</td>
<td>KolmogorovSmirnov</td>
<td>Apache Commons Math</td>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>QRDecomposition</td>
<td>JAMA library</td>
<td>6</td>
<td>72</td>
</tr>
<tr>
<td>12</td>
<td>Quadratic</td>
<td>[18, 22]</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>RootOfUnity</td>
<td>Apache Commons Math</td>
<td>6</td>
<td>27</td>
</tr>
<tr>
<td>14</td>
<td>SaddlePointExpansion</td>
<td>Apache Commons Math</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>Sort</td>
<td>SIR</td>
<td>15</td>
<td>70</td>
</tr>
<tr>
<td>16</td>
<td>Tomorrow</td>
<td>[18, 22]</td>
<td>1</td>
<td>107</td>
</tr>
<tr>
<td>17</td>
<td>TriangularDistribution</td>
<td>Apache Commons Math</td>
<td>13</td>
<td>50</td>
</tr>
</tbody>
</table>

common.validator library, which contains classes for checking the validity of input data. The class Sort was extracted from the Software artifact Infrastructure Repository (SIR) [17] and it implements different sorting algorithms, such as insertion sort and quick sort. Quadratic and Tomorrow are two classes widely used in previous studies on evolutionary test case generation [18, 22]. Finally, LU Decomposition and QR Decomposition were extracted from the Java Matrix (JAMA) library, which provides a suite of classes for performing linear algebra operations, while Arrays was extracted from the well-known Java Collections framework.

Whenever necessary, minor manual adjustments were performed on the classes for the purpose of the study, making sure that the classes are not significantly changed. Most of the modifications are superficial and they concern methods with non-numeric parameters, which are currently not supported by our prototype tool. In such cases, either a public wrapper method is added in which we pass a fixed value for the non-numeric parameters, or we simply drop the method if it is trivial (for e.g. a setter method).

5.2 Research Questions

In the context of our study, we formulated the following research questions:

- **RQ1**: Does orthogonal and/or reactive exploration improve the effectiveness of evolutionary test case generation?

This is the main research question of our study. Specifically, we aim at evaluating to what extent the orthogonal and/or reactive exploration mechanisms proposed in this paper improve the effectiveness (coverage) of GA-based test case generation.

- **RQ2**: Does orthogonal and/or reactive exploration improve the efficiency of evolutionary test case generation?

With this second research question, we aim to analyze to what extent the proposed approaches are able to reduce the cost required for reaching the highest coverage.

Note that we analyze the efficiency of the proposed approach only when we do not observe a significant improvement in terms of effectiveness. This is because during evolutionary test case generation a budget (maximum cost) is fixed a priori. Thus, every test suite generated within such a budget is acceptable from the point of view of the efficiency. This means that efficiency plays a secondary role compared to the effectiveness. In other words, a solution with a better effectiveness and a higher cost (within the available budget) is preferred to another solution with a reduced cost but worse effectiveness. On the other hand, when the effectiveness (coverage) is the same, the technique with highest efficiency (lower execution time) is preferable.

In order to answer our research questions, we compared test case generation based on St-GA with the following variants (defined in Section 3):

- **GA with history aware orthogonal exploration (H-OV-GA)**: St-GA enriched with the orthogonal exploration mechanism based on SVD and spanning the history of populations generated over time.

- **GA with reactive exploration (R-GA)**: St-GA enriched with the reactive exploration mechanism based on stagnation analysis (i.e., lack of evolution direction).

- **GA with both orthogonal and reactive exploration (R-OV-GA)**: St-GA enriched with the combination of the previous two variants, H-OV-GA and R-GA.

5.3 Metrics and Data Analysis

The effectiveness of the different approaches (RQ1) was measured in terms of branch coverage:

\[ \text{Coverage} = \frac{\#\text{covered branches}}{\#\text{total branches to be covered}} \]

while the efficiency (RQ2) in terms of consumed search budget (number of statements executed) during the search for test data. The higher the coverage, the better the effectiveness of the test data generation technique. Vice versa, the lower the number of executed statements required to reach the best coverage, the higher the efficiency of a technique.

All the approaches have been executed 100 times on each class to account for the inherent randomness in GA. Then we collected the branch coverage and the number of statements executed to reach the best coverage for each run, within a given maximum budget limit of time/cost (see Section 5.4).

In order to provide statistical support for the results, we performed statistical tests to check whether the effectiveness (or efficiency) achieved by one of the novel approaches (e.g., R-OV-GA) is significantly better than that reached by St-GA. Specifically, we used the Wilcoxon Rank Sum test [6] to test the following null hypotheses:

- **H01**: there is no difference between the effectiveness of H-OV-GA/R-GA/R-OV-GA and St-GA.

- **H02**: there is no difference between the effectiveness of H-OV-GA/R-GA/R-OV-GA and St-GA.

For H01 (effectiveness), the dependent variable is represented by the number of branches covered by an evolutionary test data generation algorithm over the 100 independent runs. Instead, for H02 (efficiency) the dependent variable is represented by the number of executed statements. Results are intended as statistically significant at $\alpha = 0.05$. Since this requires performing three tests for each subject, we adjusted the p-values using Holm’s correction procedure [16].

Besides testing the null hypotheses, it is of practical interest to estimate the magnitude of the difference, in terms of branch coverage or consumed budget, achieved by the different methods. For
this purpose, we use the Cohen $d$ effect size [5], which measures the magnitude of the effect of the different algorithms on the dependent variables. It is defined as the difference between the means, divided by the standard deviation of the differences between the coverage (number of statements) values achieved by two GA variants (e.g., St-GA and R-OV-GA). The effect size is considered large for average (number of statements) values achieved by two GA variants divided by the standard deviation of the differences between the covariates. It is defined as the difference between the means, divided by the magnitude of the effect of the different algorithms on the dependent variables. We used in our experiments are shown below:

### 5.4 GA Parameter Settings

To set the parameters of the St-GA algorithm and of the variants investigated in this study, we conducted a preliminary set of runs, aimed at assessing the sensitivity of subjects and techniques to the parameters. The same configuration was used for all the experimented variants (St-GA, H-OV-GA, R-GA and R-OV-GA), since no major sensitivity to the parameters was observed.

With regards to subjects, for most of the parameters a consistently similar set of values was used for all subjects, since the observed sensitivity was minor. However, for two parameters (population size and SVD-frequency), some major differences were observed. Hence, we group the subjects into two and use a different value for each group. The specific values of the important parameters we used in our experiments are shown below:

- **Population size**: we choose moderate population sizes of 20 for one group of subjects and 40 for the other.
- **Initial population**: for each subjects the initial population is uniformly and randomly generated within the search space $[-2048, 2048]^N$.
- **Search budget**: we restrict the search budget to a maximum number of $10^6$ executed statements or a maximum of 30 minutes of computation time for fitness evaluations. For subjects LU decomposition and QR decomposition however, because of their complexity, we used a higher budget limit of $10^7$.
- **Crossover**: we use single point fixed crossover with probability $P_c = 0.75$.
- **Mutation**: we use a uniform mutation function with probability $P_m = 1/\text{len}$ where $\text{len}$ is the dimension of the vector representing the individual.
- **Selection function**: rank selection is used with $\text{bias} = 1.7$.
- **Elitism**: the number of individuals in the current generation that are kept alive across the next generation is 1.
- **SVD-frequency**: orthogonal sub-populations are generated by SVD every $k$ generations where $k = 1$ for one group of subjects and $k = 2$ for the other. This parameter does not apply to St-GA.
- **SVD-proportion**: the proportion of individuals in the current population to be considered for estimating the evolution directions is set to $P_{\text{SVD}} = 0.25$.
- **Historic Data**: the number of generations to be considered for estimating evolutionary stagnation is $h = 10$.

The settings have been calibrated using a trial-and-error procedure, and some of them (i.e., crossover and mutation probabilities) are values commonly reported in the literature.

---

```java
class Sort {
    public static double logBeta(double p, double q) {
        if (Double.isNaN(p) || Double.isNaN(q) || (p <= 0.0) || (q <= 0.0)) {
            return Double.NaN;
        }
        final double a = FastMath.min(p, q);
        final double b = FastMath.max(p, q);
        return logGammaMinusLogGammaSum(a, b);
    }
}
```

Figure 3: Example of unreachable branch for Sort.

### 6. ANALYSIS OF THE RESULTS

This section discusses the results of our study in light of the research questions formulated in the previous section.

#### 6.1 Analysis of the Effectiveness

Table 2 reports the mean branch coverage values (over 100 independent runs) achieved by St-GA, H-OV-GA, R-GA and R-OV-GA for all the subjects. The table reports both (i) the mean number of branches and (ii) the mean percentage of the total branches covered by each algorithm.

As we can see, in 8 out of 17 cases the mean coverage value achieved by R-OV-GA is higher then that achieved by St-GA (with an improvement ranging between 0.20% and 20%). Only on SaddlePointExpansion and TriangularDistribution the effectiveness of all the experimented approaches is comparable and no clear winner can be identified. It is worth noting that on the other classes (5 out of 17) there is no difference between the experimented methods, because all the approaches achieved 100% of branch coverage.

From the analysis of the results, it also emerges that the mechanism that combines the orthogonal exploration and the reactive exploration (R-OV-GA) often turned out to be more effective than either of the two mechanisms taken individually, i.e., H-OV-GA and R-GA. Indeed, in only 4 cases (i.e., when testing Arrays, SaddlePointExpansion, TriangularDistribution and KolmogorovSmirnovDistribution) R-GA and H-OV-GA achieved a (slightly) better coverage than R-OV-GA. However, in two of these cases all the algorithms achieved almost the same level of coverage and none of the experimented algorithms is able to achieve 100% coverage.

It is important to note that the occurrence of no or a marginal improvement in terms of coverage does not necessary indicate that the proposed techniques are not useful. Indeed, by inspecting the code of such subjects, we noted that often it is impossible to achieve a higher coverage, since the uncovered branches are not reachable. For instance, let us consider the class Beta where R-OV-GA seems to be able to improve branch coverage just by 0.1% as compared to St-GA. By inspecting the code, we noticed that this is the highest possible coverage, since the uncovered branches cannot be covered by any input. Figure 3 shows an example of unreachable branch. The private method logBeta which independently performs exactly the same control before calling the
Table 2: Average coverage values achieved by St-GA, R-OV-GA, H-OV-GA and R-GA over 100 independent runs. Values in bold face turned out to be the best against those achieved by the other algorithms for the same CUT.

<table>
<thead>
<tr>
<th>Class under test</th>
<th>St-GA Cov</th>
<th>St-GA Cov %</th>
<th>H-OV-GA Cov</th>
<th>H-OV-GA Cov %</th>
<th>R-GA Cov</th>
<th>R-GA Cov %</th>
<th>R-OV-GA Cov</th>
<th>R-OV-GA Cov %</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticUtils</td>
<td>85.81</td>
<td>86.68%</td>
<td>85.89</td>
<td>86.76%</td>
<td>85.77</td>
<td>86.64%</td>
<td>85.93</td>
<td>86.80%</td>
</tr>
<tr>
<td>Arrays</td>
<td>60.20</td>
<td>80.27%</td>
<td>62.46</td>
<td>83.28%</td>
<td>59.74</td>
<td>81.65%</td>
<td>62.02</td>
<td>82.69%</td>
</tr>
<tr>
<td>Complex</td>
<td>76.91</td>
<td>85.46%</td>
<td>76.93</td>
<td>85.48%</td>
<td>76.98</td>
<td>85.53%</td>
<td>77.00</td>
<td>85.56%</td>
</tr>
<tr>
<td>CreditCardValidator</td>
<td>30.39</td>
<td>94.97%</td>
<td>31.41</td>
<td>98.16%</td>
<td>30.43</td>
<td>94.63%</td>
<td>31.69</td>
<td>98.90%</td>
</tr>
<tr>
<td>FastMath</td>
<td>36.83</td>
<td>61.38%</td>
<td>38.82</td>
<td>64.70%</td>
<td>37.48</td>
<td>62.47%</td>
<td>40.20</td>
<td>67.00%</td>
</tr>
<tr>
<td>Fraction</td>
<td>73.39</td>
<td>58.25%</td>
<td>78.59</td>
<td>83.03%</td>
<td>78.56</td>
<td>75.40%</td>
<td>78.77</td>
<td>82.52%</td>
</tr>
<tr>
<td>KolmogorovDistribution</td>
<td>37.93</td>
<td>75.86%</td>
<td>39.07</td>
<td>78.13%</td>
<td>37.70</td>
<td>75.40%</td>
<td>37.06</td>
<td>74.13%</td>
</tr>
<tr>
<td>IPAddressValidator</td>
<td>243.00</td>
<td>100.00%</td>
<td>243.00</td>
<td>100.00%</td>
<td>243.00</td>
<td>100.00%</td>
<td>243.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>LUDecomposition</td>
<td>53.55</td>
<td>70.46%</td>
<td>57.76</td>
<td>76.12%</td>
<td>56.12</td>
<td>73.84%</td>
<td>60.93</td>
<td>80.18%</td>
</tr>
<tr>
<td>QRDecomposition</td>
<td>53.04</td>
<td>73.67%</td>
<td>61.94</td>
<td>86.03%</td>
<td>56.83</td>
<td>78.93%</td>
<td>66.64</td>
<td>92.56%</td>
</tr>
<tr>
<td>Quadratic</td>
<td>7.00</td>
<td>100.00%</td>
<td>7.00</td>
<td>100.00%</td>
<td>7.00</td>
<td>100.00%</td>
<td>7.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>RootsOfUnity</td>
<td>27.00</td>
<td>100.00%</td>
<td>27.00</td>
<td>100.00%</td>
<td>27.00</td>
<td>100.00%</td>
<td>27.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>SaddlePointExpansion</td>
<td>14.00</td>
<td>87.50%</td>
<td>14.00</td>
<td>87.50%</td>
<td>14.01</td>
<td>87.56%</td>
<td>14.00</td>
<td>87.50%</td>
</tr>
<tr>
<td>Sort</td>
<td>106.90</td>
<td>99.99%</td>
<td>106.90</td>
<td>99.99%</td>
<td>107.00</td>
<td>100.00%</td>
<td>107.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>TriangularDistribution</td>
<td>42.33</td>
<td>84.66%</td>
<td>42.57</td>
<td>85.14%</td>
<td>42.24</td>
<td>84.48%</td>
<td>42.55</td>
<td>85.10%</td>
</tr>
</tbody>
</table>

Figure 4: Comparison of coverage achieved by St-GA, H-OV-GA, OV-GA and R-GA over 100 independent runs on QRDecomposition.

private method. Hence, the method logGammaMinusLogGammaSum will never throw the NumberIsTooSmallException because, on the basis of the method visibility, the public method logBeta, which is the only caller, will never pass an invalid parameter to it.

Figure 4 shows a boxplot of the experimented methods for one of the subjects: QRDecomposition. It can be seen from the boxplot that the distribution of coverage values obtained by R-OV-GA over all the independent runs is substantially higher then the distribution achieved by all the other methods. Specifically, in more than 95% of independent runs R-OV-GA achieved a coverage value ranging between 92% and 97%. Vice versa, St-GA reached a coverage value lower than 80% in the majority of all the independent runs. H-OV-GA and R-GA turned out to be quite better than St-GA, but significantly worse than R-OV-GA.

To provide a further evidence of the benefits introduced by diversity, Figure 5 compares the best fitness values achieved by the four experimented algorithms for QRDecomposition. It can be seen how R-OV-GA, R-GA and H-OV-GA converge more quickly toward better solutions, while St-GA has a lower convergence rate and is still trapped in some local optima after 23 generation. Finally, R-OV-GA seems to be able to combine the orthogonal exploration and reactive exploration showing a better ability to find better solutions than its constituents, i.e. H-OV-GA and R-GA.

All these considerations are also supported by the statistical analysis. Table 3 reports the results of the Wilcoxon tests (after correcting the p-values using Holm’s correction) and the Cohen d effect size. As we can see, in 8 out of 17 cases R-OV-GA provides a statistically significant improvement in terms of effectiveness, while for the remaining cases there is no statistically significant difference between the experimented methods. However, in 5 out of 17 cases it is impossible to improve the coverage since St-GA (as well as the other variants) achieves 100% of coverage. This means that R-OV-GA is able to improve the coverage in 70% of cases where improvement is possible.

As for the magnitude of the improvement, when comparing R-OV-GA with St-GA the effect size is generally large and only in two cases it is medium and small, respectively. It is worth noting that in the cases where there is no statistically significant difference between St-GA and R-OV-GA, the effect size is always positive. This means that R-OV-GA never reached a coverage value lower than...
St-GA. Similar considerations can be derived from the comparison of H-OV-GA and St-GA.

When comparing St-GA and R-GA, Table 3 reveals that the reactive exploration is able to significantly improve the effectiveness of St-GA with an effect size that varies over the tested classes, ranging from very large ($d \gg 1$) to small. There are also four cases where the effect size is negative, indicating that the reactive exploration taken individually does not always improve (sometimes marginally worsens) the effectiveness of GA.

In summary, for RQ1, we can assert that the proposed mechanisms for increasing the population diversity (and reducing the stagnation probability) are able to improve the effectiveness of GA-based evolutionary test case generation. In addition, among the different variants of diversification mechanisms proposed in the paper, R-OV-GA (that combines orthogonal and reactive exploration mechanisms) turned out to be the best.

### 6.2 Analysis of the Efficiency

In cases where there was no statistically significant improvement of effectiveness, we analyzed the search budget consumed by each technique and investigated whether there is any improvement in terms of efficiency. Table 4 reports the mean number of statements executed by St-GA, R-GA, H-OV-GA and R-OV-GA for reaching the best coverage. As one can see from the table, in the majority of the cases (6 out of 7 cases) R-OV-GA reached the best mean efficiency, with a cost reduction ranging from 20% to 80% on average over 100 independent runs. This means that the combined application of orthogonal and reactive exploration achieves the same coverage as St-GA, but with a cost reduction of about 40% on average (which indicates a substantial improvement in terms of efficiency).

The only exception is represented by the class Fraction, where the cost is slightly higher.

The analysis of the results also reveals that H-OV-GA and R-GA are not able to make statistically significant improvements as compared to St-GA in terms of efficiency. Indeed, for H-OV-GA the amount of consumed budget is reduced in 5 out 7 cases, while it is increased in the remaining 2 cases. Similarly, R-GA showed a reduction in terms of consumed budget in only 4 out of 7 cases.

Figure 6 shows a boxplot of the consumed search budget by the experimented methods for one of the subjects: the class Sort. The boxplot shows that the distribution of cost values obtained by R-OV-GA over all the independent runs is quite lower than the distribution achieved by all the other methods. Specifically, the budget consumed by R-OV-GA is lower than the budget required by the other algorithms for the majority (about the 75%) of the runs.

To provide statistical support for the above considerations, Table 5 reports the results of the Wilcoxon tests (after correcting $p$-values using the Holm’s correction) and the corresponding Cohen $d$ effect size. As we can see, from the point of view of efficiency R-OV-GA outperforms St-GA in a statistically significant way in 6 out 7 cases, with an effect size that is large ($d > 0.8$) in the majority of the cases. In the remaining case, the Wilcoxon test reveals that the difference is not statistically significant, even if the corresponding effect size is marginally positive.

As expected, for H-OV-GA and R-GA there is a statistically significant difference in only few cases: 4 out of 7 cases and 2 out of 7 cases, respectively.

In summary, for RQ2, we can assert that R-OV-GA is able to obtain the same effectiveness as St-GA but with a cost reduction of about 40% on average. Among the different variants of exploration mechanisms proposed in this paper, R-OV-GA provides the best efficiency as compared to R-GA and H-OV-GA.

### 7. THREATS TO VALIDITY

This section discusses the threats that could affect the validity of the evaluation of the proposed approach. Threats to construct validity concern the relation between theory and experimentation. In order to evaluate the performance of the experimented techniques we used branch coverage and number of
reactive exploration of the search space, with the principal objective of avoiding the search being dominated by a small set of similar individuals. To this aim, we presented three improved variants of the basic GA in which we incorporated history aware orthogonal, reactive, and reactive orthogonal exploration schemes in the context of evolutionary test data generation.

Experimental studies on 17 Java classes extracted from widely used open source libraries showed strong, statistically significant improvements. The proposed diversification schemes improve over basic GA in terms of effectiveness, i.e., branch coverage achieved, and efficiency, i.e., search budget consumed. In particular, the combined application of the reactive and orthogonal exploration schemes resulted in the best overall improvement. The combined method significantly improved effectiveness in 8 of the tested classes, while it significantly improved efficiency in 6 out of 7 tested classes where the effectiveness was not significantly improved.

Based on the quite promising results achieved in this paper, we intend to address a number of remaining issues in our future work. Specifically, we plan to support orthogonal exploration of search spaces with non-numeric data types (e.g., strings) as well as an arbitrary number of method call sequences as part of our future extension. For such an extension, the non-numeric types and the method call sequences need to be mapped to a vector space in which a meaningful notion of evolution direction can be defined. An idea to proceed along these directions involves mapping a variable size sequence (of method calls) to fixed size feature vectors, which represent the frequency of occurrence of each feature in an individual. Similarly, feature vectors can be used to represent strings of arbitrary length by extracting pairwise Longest Common Substrings (LCS) from the strings into a feature set and encoding each string as a feature vector containing each feature (LCS). SVD will then be applied to the feature vectors, so as to explore orthogonal regions in the space of feature vectors.

In a similar vein, since the current approach considers vectors of fixed size, we intend to investigate an extension of our technique that is able to address vectors of variable dimensions.

Finally, the current prototype implementation is meant only to show the feasibility of the proposed approach and it addresses the issue of diversity in populations from the research perspective. However, to produce a tool that can be used by test engineers for actual testing activities, there are several enhancements that should be made. For instance, incorporating special and boundary values (e.g. null, NaN, Infinity, . . .) into the initial population would help to cover trivial error-checking branches, that remain otherwise uncovered. We plan to integrate all our future extensions and enhancements of our implementation into the publicly available tool EvoSuite.
9. REFERENCES


