LIPS vs MOSA: a Replicated Empirical Study on Automated Test Case Generation

Annibale Panichella[1], Fitsum Meshesha Kifetew[2], and Paolo Tonella[2]

[1] University of Luxembourg
[2] Fondazione Bruno Kessler, Trento, Italy
annibale.panichella@uni.lu, {kifetew,tonella}@fbk.eu

Abstract. Replication is a fundamental pillar in the construction of scientific knowledge. Test data generation for procedural programs can be tackled using a single-target or a many-objective approach. The proponents of LIPS, a novel single-target test generator, conducted a preliminary empirical study to compare their approach with MOSA, an alternative many-objective test generator. However, their empirical investigation suffers from several external and internal validity threats, does not consider complex programs with many branches and does not include any qualitative analysis to interpret the results. In this paper, we report the results of a replication of the original study designed to address its major limitations and threats to validity. The new findings draw a completely different picture on the pros and cons of single-target vs many-objective approaches to test case generation.

1 Introduction

Replications are one of the key scientific practices that allow researchers to confirm, refute or adjust the validity of previous findings. In recent years, the software engineering community has seen an increasing awareness about the importance of replications and several authors view replications as a fundamental step toward the construction of solid empirical evidence in the field [7, 12, 13].

Search based test case generation aims at automatically generating a set of input vectors that reach the desired level of adequacy (e.g., branch coverage) once they are turned into test cases and executed. While the first proposals of test generators addressed one coverage target at a time [8, 14], recent approaches consider all coverage targets at the same time and either compute an aggregate fitness function for all yet uncovered targets [5] or apply a truly many-objective search to the test generation problem [10]. A novel single-target approach has been proposed in a recent paper by Scalabrino et al. [11]. The paper includes a comparison between their test generator LIPS (Linearly Independent Path based Search) and MOSA (Many-Objective Sorting Algorithm) [10]. We find the empirical investigation very interesting, since it tries to shed some light on the pros and cons of adopting a single-target vs many-objective test generation approach. However, the core contribution of the paper is not empirical. In fact, the paper is mostly focused on the novel ideas implemented in LIPS and the empirical study is a very preliminary study, conducted on a few, small C functions. Since the research question (single-target vs many-objective test generation) behind the empirical part of the LIPS paper is a key research question in search based testing, we decided to replicate and extend the empirical study reported in the LIPS paper [11].

The replicated empirical study described in this paper addresses the main threats to validity and limitations of the original study – namely, the threats to the external validity of the results, due to the size and complexity of the sample of C functions considered in the study, the threats to the internal validity, due to the way efficiency was measured and the way the parameters of the algorithms were set, and the lack of a detailed qualitative analysis of the reasons for the reported quantitative differences. The new empirical study was designed to evaluate effectiveness, efficiency and convergence of LIPS vs MOSA. Quite surprisingly, the findings of the new study differ remarkably from the original results. The new results show an undisputed superiority of the many-objective approach in all considered dimensions. The qualitative analysis of the results shows that MOSA makes a better usage of the available search budget by avoiding its allocation to a single target. Although the dynamic allocation of the search budget to a target presumably improves over its static allocation to the targets, according to our new study the many-objective approaches, which do not perform any kind of budget allocation, converge more quickly and on average achieve higher coverage.

2 Background

This section describes the two approaches being compared, LIPS and MOSA.

Linearly Independent Path based Search (LIPS). LIPS is a single-target approach proposed by Scalabrino et al. [11] for procedural languages. It uses single-objective genetic algorithms to optimise (cover) one branch (target) at a time. The fitness function for a branch is determined by the traditional approach level and branch distance [8]. In order to cover linearly independent paths to the targets, the branch selected as target is the last uncovered branch appearing in the path of the last test case that is added to the final test suite. Such a target is updated over the generations depending on whether (i) it is covered or (ii) the search budget allocated for the single target is consumed. In turn, the search budget allocated to each target is determined dynamically, as the total remaining search budget divided by the targets that are yet uncovered and that were never selected as single coverage targets in a previous generation. In this way, infeasible or difficult targets do not consume the overall search budget, because they are allocated only a fraction of the entire search budget. Moreover, even if they are not covered in the generation cycle allocated to them, they remain still coverable in successive generations thanks to collateral coverage (i.e., coverage achieved by a test case generated for a different target) or in case some residual budget remains at the end, due to easy to cover targets considered late in the process. For completeness, Algorithm 1 reports the pseudo-code for LIPS.

Algorithm 1: LIPS

```
   Input: $B = \{b_1, \ldots, b_m\}$ the set of branches to cover in the program.
          Population size $M$
   Result: A test suite $T$
 1 begin
 2     $tc_0 \longleftarrow$ randomly generated input vector
 3     $T \longleftarrow \{tc_0\}$
 4     $worklist \longleftarrow$ uncovered branches ordered as in the path traversed by $tc_0$
 5     $target \longleftarrow$ pop last branch from $worklist$
 6     START-CLOCK-FOR-TARGET($target$)
 7     $budget \longleftarrow$ LOCAL-BUDGET()
 8     $t \longleftarrow 0$ // current generation
 9     $P_t \longleftarrow$ RANDOM-POPULATION($M - 1$) $\bigcup\{tc_0\}$ // Initial population
10     while | $worklist$ |> 0 AND not(overall search budget consumed) do
11         if $t > 0$ then
12             $P_t \longleftarrow$ GENERATE-OFFSPRING($P_{t-1}$)
13         COLLATERAL-COVERAGE($P_t$, $worklist$)
14         if $target$ is covered then
15             $T \longleftarrow T \bigcup$ {test $tc$ covering $target$}
16             $worklist \longleftarrow$ UPDATE-WORKLIST($tc$)
17             $target \longleftarrow$ pop last branch from $worklist$
18             START-CLOCK-FOR-TARGET($target$)
19         else if CLOCK-FOR-TARGET($target$) $\geq$ $budget$ then
20             $target \longleftarrow$ pop last branch from $worklist$
21             START-CLOCK-FOR-TARGET($target$)
22         $budget \longleftarrow$ LOCAL-BUDGET()
23         $t \longleftarrow t + 1$
```

It should be noted that no pseudo-code is available in the paper by Scalabrino et al.[11]. Moreover, the source code of the tool is also not available. Hence, we have elaborated the pseudo-code by trying to follow the specifications of LIPS available in the paper as strictly as possible. However, sometimes the description in the paper is not detailed enough for unambiguous interpretation and we had to make decisions on what to implement. While this might have produced differences between our and the original implementation of LIPS, we think that the key ideas behind LIPS, i.e., the ordering of the targets by execution path and the dynamic re-allocation of the search budget, are captured faithfully in our implementation. Moreover, by providing the pseudo-code for LIPS in our paper we contribute to the disambiguation of the minor, yet important, details behind the ideas described in the paper.

LIPS starts with an initial, randomly generated test case tc_0 (line 2 in Algorithm 1), which represents the input vector for the program under test [11]. Such a test is executed and the uncovered branches for all decision nodes in the execution path of tc_0 are added to a worklist (line 4 in Algorithm 1) in the order in which they are encountered. The worklist represents the queue of branches that can be potentially considered as search targets. Starting from tc_0, the genetic algorithm is initialised as follows [11]: (i) the initial search target is the last branch added to the worklist (line 5), and (ii) an initial population that includes tc_0 is randomly generated (line 9). In the evolutionary iterations, new tests are generated using crossover and mutation (GENERATE-OFFSPRING at line 12). Parents are selected using the tournament selection and according to the single fitness function of the current target [11]. Whenever a newly generated test tc covers the current target, (i) it is added to the test suite (at line 15), (ii) all the uncovered branches of decision nodes on the path covered by tc are added to the worklist in the order in which they are encountered (UPDATE-WORKLIST at line 16). If some of the uncovered targets in the path of tc were selected before as coverage targets, they are added at the front of the worklist, so that they are selected as current coverage targets only when all the other targets, which were never tried before, are covered, using the residual (if any) search budget. The last branch added to the worklist is selected as new target (line 17). The branches in the worklist that are covered (by chance) by the newly generated tests are removed from the worklist and marked as "covered" (COLLATERAL-COVERAGE at line 13). Since LIPS targets one branch at a time, it has to allocate a portion of the overall search budget for each uncovered and not previously selected branch. To account for collateral coverage, which could free some search budget, at each generation the budget is re-computed as SB/n, where SB is the budget that remained available after last target selection and n is the number of remaining uncovered branches that were never selected before (LOCAL-BUDGET at line 22). If the current target is not covered within the allocated *budget*, a new target is selected from the worklist (lines 19-20). The main loop at lines 10-23 is repeated until all the branches are covered or the total search budget is consumed [11].

LIPS has been defined for procedural programs written in C. Therefore, it does not address the problem of generating method sequences [14], which means it is not directly applicable to object-oriented programs. Moreover, the length of the chromosome used by LIPS is fixed, which means that data structures with variable size (e.g., arrays) are assigned a predefined, fixed size. This may prevent coverage of targets requiring a specific, special value of size (e.g., a condition that checks if an array has size zero). Finally, although not stated explicitly in the paper [11], we assume that LIPS uses elitism in GENERATE-OFFSPRING (line 11), given the fact the elitism has been shown to positively affect the convergence speed of GAs in various optimisation problems and it is also used (although in a different way) in MOSA.

Many-Objective Sorting Algorithm (MOSA). MOSA is a many-objective genetic algorithm proposed by Panichella et al. [10] for Java classes and implemented in EvoSuite[3]. A test case in MOSA is a method sequence (including input data) of variable length, which is evaluated against all uncovered branches. MOSA targets all uncovered branches at once by considering them as different (many) objectives to be optimised in parallel. It shares the same main loop with NSGA-II [4], which is one of the most popular multi-objective genetic algorithms. However, it differs on three key aspects: (i) it selects test cases according to a preference criterion suitably defined for the test case generation problem; (ii) it considers as objectives only the yet uncovered coverage branches (i.e., the set of optimisation objectives changes across generations); (iii) it uses an archive to store all test cases satisfying one or more previously uncovered branches. The pseudo-code of MOSA is shown in Algorithm 2 [10].

MOSA starts with an initial set of randomly generated test cases (line 3 of Algorithm 2); then, new test cases (*offspring*) are created using *crossover* and *mutation* (GENERATE-OFFSPRING, at line 6 of Algorithm 2). Then, parents and offspring are selected to form the next generation according to their ranks,

[3] `https://github.com/EvoSuite/evosuite/tree/master/client/src/main/java/org/evosuite/ga/metaheuristics/mosa`

Algorithm 2: MOSA

```
    Input: $B = \{b_1, \ldots, b_m\}$ the set of coverage targets of a program.
           Population size $M$
    Result: A test suite $T$
1   begin
2       $t \longleftarrow 0$ // current generation
3       $P_t \longleftarrow$ RANDOM-POPULATION($M$)
4       archive $\longleftarrow$ UPDATE-ARCHIVE($P_t$, $\emptyset$)
5       while not (search budget consumed) do
6           $Q_t \longleftarrow$ GENERATE-OFFSPRING($P_t$)
7           archive $\longleftarrow$ UPDATE-ARCHIVE($Q_t$, archive)
8           $R_t \longleftarrow P_t \bigcup Q_t$
9           $\mathbb{F} \longleftarrow$ PREFERENCE-SORTING($R_t$)
10          $P_{t+1} \longleftarrow \emptyset$
11          $d \longleftarrow 0$
12          while $\mid P_{t+1} \mid + \mid \mathbb{F}_d \mid \leqslant M$ do
13              CROWDING-DISTANCE-ASSIGNMENT($\mathbb{F}_d$)
14              $P_{t+1} \longleftarrow P_{t+1} \bigcup \mathbb{F}_d$
15              $d \longleftarrow d + 1$
16          SORT($\mathbb{F}_d$) //according to the crowding distance
17          $P_{t+1} \longleftarrow P_{t+1} \bigcup \mathbb{F}_d[1 : (M- \mid P_{t+1} \mid)]$
18          $t \longleftarrow t + 1$
19      $T \longleftarrow$ archive
```

as determined by the PREFERENCE-SORTING routine [10] (line 9). Tests that satisfy the preference criterion are assigned to the first front $\mathbb{F}_0$ while all the remaining tests are ranked using the non-dominated sorting algorithm of NSGA-II [4]. The preference criterion prioritises test cases that are closer to one or more uncovered branches (according to the corresponding branch distance and approach level scores). When there are multiple test cases with the same objective scores, the preference criterion uses the test case length as secondary selection criterion [10], i.e., shorter tests are preferred. The population for the next generation is formed using the loop at lines 12-15: test cases are selected starting from those in front $\mathbb{F}_0$, then those in front $\mathbb{F}_1$, and so on. At the end of the loop (lines 16-17), the remaining test cases are selected from the current front $\mathbb{F}_d$ according to the descending order of crowding distance. Finally, MOSA uses an *archive*, to keep track of the shorter test cases that cover the branches of the program under test. Whenever new test cases are generated (either at the beginning of the search or when creating offspring), MOSA stores those tests that cover previously uncovered targets in the *archive* as candidates to form the final test suite (function UPDATE-ARCHIVE at lines 4 and 7).

MOSA has been defined for Java classes. Therefore, it addresses both test data and method sequence generation. Since it is implemented in EvoSuite, it can handle complex data structures as input, such as objects and arrays of objects. The encoding schema (the standard one in EvoSuite [6]) allows to create test cases (i.e., method sequences) as well as test inputs (e.g., arrays) with variable length. Finally, MOSA uses elitism: test cases closer to satisfying uncovered branches (or with minimum length at the same level of "closeness") are guaranteed to survive in the next generation [10].

3 Summary of the replicated empirical study

This section summarises the empirical study published by Scalabrino et al. [11] comparing LIPS and their reimplementation of MOSA when generating test inputs for C functions. The empirical evaluation was performed on 35 C functions with number of branches ranging between 2 and 64 (15 branches per function on average). These C functions are taken from different open-source C libraries [11]: (i) 21 functions from `gimp`, an open source GNU image manipulation software; (ii) five functions from `GSL`, the GNU Scientific Library; (iii) three functions from *SGLIB*, a generic library for C; (iv) three functions from `spice`, an analogue circuit simulator; (v) one function from `bibclean`; and (vi) two functions from previous work on test data generation for the C language.

The comparison is performed on three different dimensions: (i) branch coverage (*effectiveness*), (ii) execution time (*efficiency*), (iii) number of tests in the final test suite (*oracle cost*). According to the results of the study, there is no difference in terms of branch coverage between LIPS and MOSA for the majority of the C functions. In ten out of 35 functions LIPS has a better branch coverage than MOSA and for these cases the average difference is 5.72%. In two out of 35 cases MOSA outperforms LIPS and the average difference in branch coverage for these cases is 5.61%. Notice that these average values are obtained from Table 2 of the LIPS paper [11]. LIPS is reported to be more efficient than MOSA for all C functions, with an average improvement around 66% in terms or running time. Scalabrino et al. [11] measure efficiency as the execution time required to perform 200,000 fitness function evaluations for the 27 functions with less than 100% coverage. Finally, they report that MOSA produces significantly shorter test suites compared to LIPS in 32 out of 35 functions. However, the differences are easily ironed out by greedy algorithms for test suite minimisation [11]. As reported in [11], the execution time for the minimisation is negligible given the small size of the functions under test and of the generated test suites.

3.1 Threats to validity

We have identified the following threats to the validity of the original study and we believe that a replication of the study is very important to address them.

Threats to **external validity** affect the generalisation of the results. Among them, the number and size/complexity of the considered C functions affect the external validity of the reported findings to a major extent. *Size/complexity of the functions*: the selected functions are small and contain few branches. Only two functions have more than 50 branches (i.e., 56 and 64 branches) and 16 out of 35 functions (46%) have less than ten branches each. For comparison, MOSA was originally evaluated on Java classes with at least 50 branches each [10]. Therefore, it is not clear to what extent results are generalisable to functions with more than 50 branches. *Number of functions*: the empirical study considers only 35 small C functions. A larger sample is needed to extend the validity of the findings of the study.

Threats to **internal validity** regard internal factors that could have influenced the experimental results. Among them, the measurement of efficiency and the setting of some critical parameters of the algorithms might have affected the internal validity of the study. *Measurement of efficiency*: efficiency is measured as the execution time required by each algorithm to run until 200,000 fitness function evaluations are made and not as the execution time needed to reach maximum coverage. The chosen setting favours LIPS by design, since each generation of MOSA is more expensive to compute, due to the cost of the ranking procedure. Therefore, it is not clear whether the execution time needed by the two algorithms to reach maximum coverage differs or not.

Moreover, looking at the results reported in the original study in Table 2, we observed some inconsistencies in the execution time between LIPS and MOSA for simple C functions, where 100% of coverage is reached. For example, for function `gimp_hsl_value_int` LIPS required less than 10 milliseconds to reach 100% coverage. Since this function is very simple, it can be presumably covered fully in the first generation (i.e., with no need for evolution). However, for MOSA the reported running time to reach full coverage on the same function is 10.23s. If the initial populations for MOSA and LIPS are the same (i.e., randomly generated), both algorithms should achieve full coverage within approximately the same time. We observed the same inconsistency in seven other very simple C functions used in the study [11].

3.2 Reasons to replicate

In addition to the possibility of addressing some of the threats to the validity of the original study, there are further reasons for replicating the empirical study by Scalabrino et al. [11]. First, the study provides only a quantitative analysis of the collected results, without attempting to interpret them qualitatively. An in-depth qualitative analysis would allow us to better understand under which conditions one algorithm outperforms the other. Moreover, a further study is needed to better understand how LIPS and MOSA perform on programs with a large number of branches (> 50). In fact, a recent study [9] involving classes with both low and high number of branches confirmed the higher effectiveness and efficiency of MOSA (and its improved variant DynaMOSA) for classes with high number of branches (high cyclomatic complexity) [9]. On the other hand, Scalabrino et al. [11] compared LIPS and MOSA on C functions with less than 20 branches on average (only one function has slightly more than 50 branches). Hence, there is a strong need for a larger study, with both small (< 50 branches) and large (≥ 50 branches) programs.

3.3 How to replicate

In principle, the simplest option to replicate the study would be to re-run LIPS and MOSA on a larger sample of C functions using OCELOT, i.e., the tool that implements LIPS [11]. However, this option is not viable since OCELOT and the code for LIPS are not publicly available at the time of this submission.

The viable alternative is to re-implement LIPS in EvoSuite. Differently from OCELOT, EvoSuite is publicly available on GitHub[4] and it already contains the original code of MOSA [10]. An important drawback of this choice is that EvoSuite generates test suites for Java classes and not for C functions. Therefore, this

[4] `https://github.com/EvoSuite/evosuite`

option requires the conversion of the C functions used in the original study [11] into Java static methods. Fortunately, this conversion is straightforward since the selected C functions do not have complex input parameters with advanced C syntax (e.g., pointers to complex structures). Since EvoSuite supports the generation of test cases and input data (e.g., arrays) of variable length, as a side effect of using EvoSuite we also overcome one of the limitations of LIPS/OCELOT: the fixed chromosome size, discussed in Section 2.

4 Design of the new study

We first describe our re-implementation of LIPS within EvoSuite. Then we describe the selected subjects, the research questions and the metrics we adopt to answer them.

Implementation of LIPS in EvoSuite: We have re-implemented LIPS based on the pseudo-code in Algorithm 1, within EvoSuite version `1.0.5`, available from GitHub on March 12th, 2017. The main differences between the original version of LIPS [11] and our re-implementation regard the encoding schema and the genetic operators, for which we use the default settings in EvoSuite [5]. In EvoSuite [5], a test case is a sequence of statements, which is composed of method calls (i.e., call to static methods) and data inputs (e.g., arrays, strings). New test cases are generated by applying *single-point* crossover and *uniform mutation*. The latter can remove, change, or add statements from/in/to the test cases. While we allow the length of the test cases to vary during the GA search, so that the length of input arrays, strings, etc., can change, we only allow one method execution for the class under test, i.e., the execution of the static method under test, because we are interested in evaluating LIPS vs. MOSA for procedural, stateless methods only. Selection is *tournament selection*, the same operator originally proposed for LIPS [11]. The encoding schema and genetic operators are the same for MOSA [10, 9], i.e., they work at test case level.

In the original LIPS implementation [11], the length of the chromosomes is fixed a priori, which might prevent coverage of specific branches. In addition, the original genetic operators [11] are *blend-crossover* (BLX) and *polynomial mutation*, which can be applied only to chromosomes with fixed length and containing only numerical values [3]. Since our re-implementation of LIPS in EvoSuite does not have such constraints, we deem it as superior to the original implementation and eventually able to cover more branches. We found this conjecture to be empirically true by comparing the results of our re-implementation with the results reported in the original study, considering the common subset of programs under test (see Section 5). It should be noticed that the core novelties of LIPS, namely the order by which branches are selected as targets and the dynamic allocation of the search budget, are kept identical to the original formulation in our re-implementation. Our re-implementation of LIPS is publicly available for download on GitHub: `https://github.com/apanichella/evosuite/tree/LIPS_replication`.

Benchmark: Since LIPS was originally defined for procedural functions and not for object oriented programs, in our replication study we target only static methods with purely procedural behaviour. Our benchmark contains 70 static methods characterised as follows: (i) 33 static methods are the Java equivalent of the C functions used in the original study [11]; (ii) 37 additional static methods have been randomly selected from Java open-source libraries. Notice that we excluded two of the 35 functions used in [11], namely `Csqrt` and `triangle`, for which we could not find the source code. Our benchmark contains twice as many subjects as the original study [11]. Moreover, 14 subjects have more than 50 branches each, thus allowing to compare LIPS and MOSA on very large/complex functions. In general, the number of branches[5] in each static method ranges between 3 and 425. The characteristics of the Java static methods (i.e., name and number of branches) are detailed in Table 1.

Porting the old benchmark to Java. All C functions used in the original study take as input primitive data types, pointers to primitive data types and arrays. Therefore, porting such functions to Java was straightforward: for each function `f`, we create a corresponding Java class containing only one single static method with the same content and the same parameters of `f`.

New subjects. To increase the size of the benchmark, we randomly selected 37 Java static methods from seven open-source libraries. In particular, we selected: (i) 17 methods from the `apache commons math` (*math* in Table 1); (ii) seven from `apache commons lang` (*lang*); (iii) two from `apache commons io` (*io*); (iv) three from `joptimizer`[6] (*IOpt.*); (v) two from `nd4j`[7] (*nd4j*); (vi) one from `google gson` (*Gson*); (vii) three from `apache commons imaging` (*imaging*) (viii) two from `apache commons bcel` (*Bcel*).

4.1 Research questions and performance metrics

We investigate the following research questions:

- **RQ1**: *How do LIPS and MOSA perform in terms of effectiveness?*
- **RQ2**: *How do LIPS and MOSA perform in terms of efficiency?*
- **RQ3**: *Does the program size (number of branches) affect the performance of LIPS and MOSA?*

To answer **RQ1**, we use the same measure of *effectiveness* used in the original study, i.e., the percentage of covered branches. For the efficiency (**RQ2**), we do not use the measure used by Scalabrino et al. [11]. This is because, as explained in Section 3, the execution time required by each approach to perform 200,000 fitness function evaluations penalises by design MOSA and does not consider the time actually needed to reach maximum coverage, independently of the number

[5] The number of branches reported here is sometimes slightly different from that of the original study because EvoSuite performs the instrumentation and counts thebranches at the byte code, not source code, level.

[6] `http://www.joptimizer.com`

[7] `http://nd4j.org`

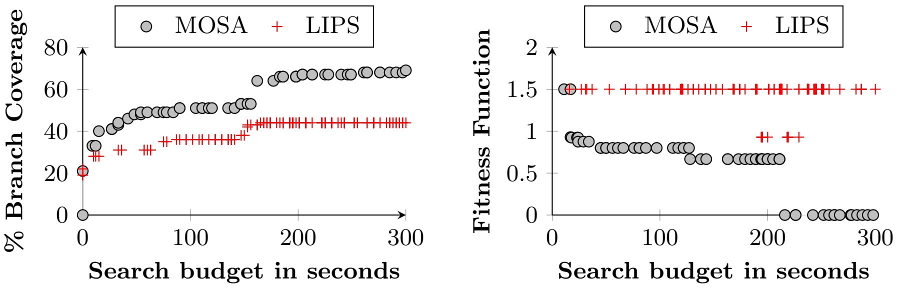
of fitness evaluations consumed to reach it. Instead, we use an overall maximum allowed execution time as stop condition, i.e., the two approaches are executed for the same amount of time (if full coverage is not reached; otherwise execution stops earlier). Then, we measure the *efficiency* as the execution time required by each approach to reach maximum branch coverage. Moreover, we consider *efficiency* as a secondary performance metric: we compare LIPS and MOSA in terms of efficiency only for those subjects with no statistically significant difference in effectiveness. Notice that we do not compare the length of the test cases since EvoSuite applies test minimisation by default.

For each subject, each search approach (LIPS or MOSA) is run 50 times to address the random nature of the genetic algorithms. In each run, we collect the percentage of covered branches (**RQ1**) as well as the elapsed time between the start of the search and the latest increment in branch coverage (**RQ2**). We report the average coverage and execution time achieved by LIPS and MOSA over these independent runs. To provide statistical support to the analysis of the results, we apply the non-parametric Wilcoxon Rank Sum test [2] with a significance level of $\alpha = 0.05$. We also measure the effect size (i.e., the magnitude) of the differences (if any) in effectiveness or efficiency using the Vargha-Delaney ($\hat{A}_{12}$) statistic [15]. Finally, to answer **RQ3**, we use the one-way permutation test [1] to verify whether there is any significant interaction between effectiveness/efficiency of the two approaches on one side and complexity of the static method under test, measured as the number of branches to cover, on the other side. In particular, we use the number of branches in the methods as independent variable and the $\hat{A}_{12}$ statistics (obtained from the comparison) as dependent variable. We set the test with a significance level of $\alpha = 0.05$ and a number of iterations equal to 10^8 (a number of iterations $> 1,000$ is recommended for this test [1]). The one-way permutation test is a non-parametric test, thus, it does not make any assumption on data distributions.

Parameter setting. We adopted the default parameter values used by EvoSuite[5] for both LIPS and MOSA, with the only exception of those parameters explicitly mentioned in the original study [11]. Therefore, we set the population size to 100 individuals and the crossover probability to 0.90. For the search budget, we fix the same maximum execution time of one minute for both LIPS and MOSA. This value (60 seconds) corresponds to the largest running time observed in the original study. Therefore, LIPS and MOSA terminate either when 100% of branch coverage is reached or when the maximum search budget of one minute is consumed.

5 Experimental results

Table 1 reports the mean branch coverage (**RQ1**) and mean execution time (**RQ2**) achieved by LIPS and MOSA for each Java static method over 50 independent runs. The table also reports the *p*-values of the Wilcoxon test as well as the corresponding $\hat{A}_{12}$ statistics (effect size). Notice that values of $\hat{A}_{12} > 0.5$ indicate that LIPS is more effective (higher branch coverage) or more efficient



(a) Percentage of covered branches over search time

(b) Fitness values for one of the branches covered by MOSA but not by LIPS

Fig. 1. Comparison between LIPS and MOSA with a larger search budget of five minutes for method `BasicCParser.preprocess`

(lower execution time) than MOSA; values of $\hat{A}_{12} < 0.5$ indicate that MOSA is more effective or more efficient than LIPS.

Results for RQ1. From columns 4-7 of Table 1, we can observe that in 45 out of 70 subjects (64%) there is no statistically significant difference in terms of branch coverage between LIPS and MOSA. Among these 45 subjects, 26 (60%) are trivial subjects that are fully covered in few seconds and 20 come from the original study [11]. In none of the remaining subjects LIPS could outperform MOSA in terms of branch coverage. Instead, MOSA achieves statistically significantly higher branch coverage than LIPS in 25 out of 70 subjects (36%). In these cases, the average (mean) difference in branch coverage is 7.67%, with a minimum of 0.92% and a maximum of 22.94%. The subject with the largest difference is `NumberUtils.createNumber` from `apache commons math`, which contains 115 branches. For this method, LIPS achieved 65.43% branch coverage compared to 88.36% achieved by MOSA (+26 covered branches) within one minute.

To better understand whether the observed differences vary when increasing the search budget, Figure 1-(a) shows the average branch coverage achieved by LIPS and MOSA over a larger search budget of five minutes for method `BasicCParser.preprocess` from `apache common imaging`. In the first generation (i.e., at time zero), the two approaches have the same average coverage since they both start with a randomly generated population. However, after the first 20s the scenario dramatically changes: MOSA yields a higher coverage for the rest of the search, leading to a difference of +25% at the end of the search. Figure 1-(b) depicts the fitness function values for the false branch b_{30} of the statement `if (c==‘\r’ || c==‘\n’)` at line 196 of class `BasicCParser` and placed inside multiple `if` statements within a `for` loop. MOSA takes around 205s to cover b_{30}, although this is one of the targets since the beginning of the search. Instead, LIPS selects this branch as its current target after 236s and only for 3s in total, which is not enough to cover it. Moreover, the fitness function curve is not monotonic in LIPS: it decreases between 194s and 200s but it increases in the next generations since b_{30} is not yet considered as the current target. A similar trend can be observed between 218s and 229s. Instead, in

Table 1. Average (mean) results for **RQ1 (effectiveness)** and **RQ2 (efficiency)**

Project	**Method/Function Name**	**Tot.**	**% Branch Coverage**				**Execution Time (ms)**			
		Branches	**LIPS**	**MOSA**	***p*-value**	$\hat{\mathbf{A}}_{12}$	**LIPS**	**MOSA**	***p*-value**	$\hat{\mathbf{A}}_{12}$
bibclean	check_ISBN	47	89.36	89.66	0.16	0.48	511	**313**	**<0.01**	**0.10**
gimp	gimp_cmyk_to_rgb_int	3	100.00	100.00	1.00	0.50	248	**217**	**<0.01**	**0.15**
gimp	gimp_cmyk_to_rgb	7	98.86	100.00	0.16	0.48	348	**320**	**<0.01**	**0.29**
gimp	gimp_gradient_calc_bilinear_factor	9	94.00	**100.00**	**<0.01**	**0.00**				
gimp	gimp_gradient_calc_conical_asym_factor	9	100.00	100.00	1.00	0.50	500	**295**	**<0.01**	**0.00**
gimp	gimp_gradient_calc_conical_sym_factor	11	99.42	100.00	0.15	0.48	525	**311**	**<0.01**	**0.00**
gimp	gimp_gradient_calc_linear_factor	13	100.00	100.00	1.00	0.50	884	**547**	**<0.01**	**0.24**
gimp	gimp_gradient_calc_radial_factor	9	88.89	88.89	1.00	0.50	477	**289**	**<0.01**	**0.09**
gimp	gimp_gradient_calc_spiral_factor	11	100.00	100.00	1.00	0.50	552	**289**	**<0.01**	**0.02**
gimp	gimp_gradient_calc_square_factor	9	88.89	88.89	1.00	0.50	551	**287**	**<0.01**	**0.04**
gimp	gimp_hsl_to_rgb_int	5	100.00	100.00	1.00	0.50	478	**236**	**<0.01**	**0.00**
gimp	gimp_hsl_to_rgb	9	98.89	**100.00**	**0.02**	**0.45**				
gimp	gimp_hsl_value_int	11	100.00	100.00	1.00	0.50	501	**260**	**<0.01**	**0.00**
gimp	gimp_hsl_value	11	100.00	100.00	1.00	0.50	415	**269**	**<0.01**	**0.05**
gimp	gimp_hsv_to_rgb	23	99.83	100.00	0.16	0.48	19773	**6223**	**<0.01**	**0.03**
gimp	gimp_rgb_to_cmyk	13	99.08	**100.00**	**0.04**	**0.46**				
gimp	gimp_rgb_to_hsl_int	15	100.00	100.00	1.00	0.50	2167	**1231**	**<0.01**	**0.23**
gimp	gimp_rgb_to_hsl	17	93.77	94.12	0.16	0.48	791	**339**	**<0.01**	**0.01**
gimp	gimp_rgb_to_hsv_int	17	94.12	94.12	1.00	0.50	958	**455**	**<0.01**	**0.10**
gimp	gimp_rgb_to_hsv4	17	86.00	**88.24**	**<0.01**	**0.31**				
gimp	gimp_rgb_to_hwb	11	100.00	100.00	1.00	0.50	623	**312**	**<0.01**	**0.00**
gimp	gimp_rgb_to_l_int	3	100.00	100.00	1.00	0.50	341	**166**	**<0.01**	**0.00**
gsl	gsl_poly_complex_solve_cubic	23	86.96	86.96	1.00	0.50	623	350	**<0.01**	0.03
gsl	gsl_poly_complex_solve_quadratic	15	100.00	100.00	1.00	0.50	528	**393**	**<0.01**	**0.06**
gsl	gsl_poly_eval_derivs	13	100.00	100.00	1.00	0.50	1995	**1472**	**0.07**	**0.40**
gsl	gsl_poly_solve_cubic	21	85.71	85.71	1.00	0.50	723	**353**	**<0.01**	**0.01**
gsl	gsl_poly_solve_quadratic	15	100.00	100.00	1.00	0.50	406	**357**	**0.01**	**0.34**
sglib	sglib_int_array_binary_search	11	100.00	100.00	1.00	0.50	424	**367**	**<0.01**	**0.31**
sglib	sglib_int_array_heap_sort	18	100.00	100.00	1.00	0.50	1963	**1222**	**<0.01**	**0.27**
sglib	sglib_int_array_quick_sort	37	97.30	97.30	1.00	0.50	3778	**2467**	**<0.01**	**0.25**
spice	clip_line	47	78.47	**80.21**	**<0.01**	**0.32**				
spice	clip_to_circle	57	93.54	**96.18**	**<0.01**	**0.12**				
spice	cliparc	95	97.41	**98.95**	**<0.01**	**0.10**				
math	ArithmeticUtils.gcd	29	96.55	96.55	1.00	0.50	1410	**439**	**<0.01**	**0.05**
math	ArithmeticUtils.mulAndCheck	17	100.00	100.00	1.00	0.50	538	**383**	**<0.01**	**0.01**
math	CombinatoricsUtils.binomialCoefficient	21	100.00	100.00	1.00	0.50	3202	**1845**	**<0.01**	**0.21**
math	CombinatoricsUtils.binomialCoefficientLong	19	100.00	100.00	1.00	0.50	2588	**1818**	**<0.01**	**0.31**
math	CombinatoricsUtils.strirlingS2	29	86.14	**96.55**	**<0.01**	**0.36**				
math	MathArrays.checkOrder	25	96.00	96.00	1.00	0.50	380	**337**	**<0.01**	**0.30**
math	MathArrays.isMonotonic	21	95.14	95.24	0.33	0.49	400	384	0.22	0.43
math	MathArrays.safeNorm	21	100.00	100.00	1.00	0.50	9346	**3931**	**<0.01**	**0.29**
math	MathArrays.shuffle	15	93.33	93.33	1.00	0.50	567	517	0.60	0.47
math	MathArrays.sortInPlace	26	92.31	92.31	1.00	0.50	**1357**	2964	**<0.01**	**0.79**
math	MedianOf3PivotingStrategy	11	81.64	**96.18**	**<0.01**	**0.34**				
math	OpenIntToDoubleHashMap.findInsertionIndex	23	94.17	**99.30**	**<0.01**	**0.28**				
math	OpenIntToFieldHashMap.findInsertionIndex	23	94.00	**99.30**	**<0.01**	**0.22**				
math	FastMath.scalb	41	93.85	**97.51**	**<0.01**	**0.26**				
math	FastMath.exp	25	97.28	**98.64**	**<0.01**	**0.34**				
math	FastMath.atan	19	100.00	100.00	1.00	0.50	638	800	0.07	0.61
math	FastMath.atan2	69	74.58	**81.16**	**<0.01**	**0.03**				
lang	NumberUtils.isCreatable	121	71.97	**89.69**	**<0.01**	**0.00**				
lang	NumberUtils.createNumber	115	65.43	**88.37**	**<0.01**	**0.00**				
lang	Fraction.greatestCommonDivisor	33	96.97	96.97	1.00	0.50	697	**370**	**<0.01**	**0.25**
lang	RandomStringUtils.random	53	81.40	**88.30**	**<0.01**	**0.12**				
lang	DurationFormatUtils.formatPeriod	47	87.15	**90.43**	**<0.01**	**0.08**				
lang	DateUtils.modify	71	1.41	1.41	1.00	0.50	460	481	0.62	0.53
lang	WordUtils.wrap	27	100.00	100.00	1.00	0.50	1965	2547	0.93	0.51
IO	FilenameUtils.getPrefixLength	51	88.71	**99.84**	**<0.01**	**0.02**				
IO	FilenameUtils.wildcardMatch	37	93.19	**96.86**	**<0.01**	**0.22**				
JOpt.	ColtUtils.squareMatrixInverse	3	100.00	100.00	1.00	0.50	999	1042	0.70	0.48
JOpt.	ColtUtils.getMatrixScalingFactors	51	81.57	84.31	0.16	0.48	1056	1013	0.30	0.44
JOpt.	ColtUtils.calculateDeterminant	19	93.16	96.95	0.80	0.49	8576	9742	0.95	0.50
nd4j	BigDecimalMath.Gamma	15	94.13	92.78	0.10	0.59	32969	29776	0.21	0.43
nd4j	BigDecimalMath.zeta	21	100.00	100.00	1.00	0.50	23734	**16530**	**<0.01**	**0.30**
Gson	ISO8601Utils.parse	83	21.01	**33.86**	**<0.01**	**0.03**				
Imaging	BasicCParser.preprocess	109	31.63	**45.91**	**<0.01**	**0.14**				
Imaging	BasicCParser.unescapeString	71	22.11	**36.48**	**0.02**	**0.36**				
Imaging	T4AndT6Compression.compressT4_2D	39	100.00	100.00	1.00	0.50	4111	3251	0.17	0.37
Bcel	Utility.signatureToString	83	74.41	**80.75**	**<0.01**	**0.05**				
Bcel	Utility.codeToString	425	73.13	**88.34**	**<0.01**	**0.00**				
	Average (mean) results		89.18	92.06			7217	5380		

MOSA the fitness function curve is monotonic because the best test case for b_{30} is preserved (elitism) until a better test is found in the subsequent generations. Instead, in LIPS (as well as in any other single objective genetic algorithm), elitism holds only for the single fitness function being optimised (i.e., only for the current target).

Results for RQ2. For the 45 methods with no statistically significant difference in effectiveness, we compare the execution time required by LIPS and

MOSA to achieve the highest coverage. The results of this comparison are reported in columns 8-11 of Table 1. Out of 45 methods, LIPS is significantly more efficient than MOSA in only one method, i.e., `MathArrays.sortInPlace`. For this method, LIPS required 1.36s on average to reach a coverage of 92% while MOSA spent 2.96s on average to reach the same branch coverage. On the other hand, MOSA is significantly more efficient than LIPS in 33 methods (73%). The minimum (yet significant) difference of 0.28s is observed for `gimp_cmyk_to_rgb` while the maximum of 13.55s is observed for `gimp_hsv_to_rgb`. For the remaining 11 methods, there is no significant difference between LIPS and MOSA.

Results for RQ3. For what concerns coverage, the one-way permutation test reveals that the $\hat{A}_{12}$ statistics is significantly influenced by the number of branches of the function/method under test (p-value < 0.01). In other words, MOSA achieves significantly higher branch coverage over LIPS especially for methods with high number of branches. For the execution time, the one-way permutation test reveals a marginally significant interaction between $\hat{A}_{12}$ statistics and the number of branches (p-value=0.06). Thus, we can conclude that the size/complexity of the program under test affects the performance (coverage and execution times) of LIPS and MOSA: the former approach is less scalable than the latter when the number of branches to cover increases.

5.1 Comparison between old and new results

We draw completely different conclusions from our results with respect to the original study. The main differences and observations are summarised below.

Superiority of our re-implementation of LIPS. For the 33 subjects shared with the original study, we observe that our re-implementation of LIPS could achieve 100% of coverage for 15 methods within 0.80s on average. Instead, in the original study LIPS reached 100% of coverage in only 8 cases [11]. This highlights the superiority of our re-implementation in EvoSuite compared to the original implementation, confirming our theoretical observations in Section 4. For example, for function `gimp_rgb_to_hwb` the original LIPS implementation reached only 50% coverage in 7.97s [11]. Instead, LIPS re-implemented in EvoSuite achieved 100% coverage in 0.62s.

MOSA is more effective than LIPS. Despite these improvements, LIPS could never achieve significantly higher coverage than MOSA. Instead, MOSA achieved significantly higher coverage on 36% of the subjects. To understand these results, let us consider `Utility.codeToString`, which has 425 branches. Given the high number of branches, LIPS can allocate a limited search budget to each branch, even in the presence of dynamic budget reallocation. As a consequence, LIPS can cover only the trivial branches that do not need many generations of test evolution. Instead, MOSA evolves test cases targeting all the branches at the same time, for the whole duration of the search budget.

MOSA is more efficient than LIPS. Our results contradict the results of the original study in terms of efficiency [11]. The main reason for such different conclusions is the different stop condition considered in the two studies: time required to perform 200,000 fitness evaluations (original study) vs. time needed to reach the same final coverage (new study). We believe the new stop condition provides a fair measurement of the respective time performance of the two algorithms, since 200,000 fitness evaluations are not necessarily required by both algorithms to reach the final coverage – indeed, they typically need a different number of fitness evaluations.

Most subjects in the original study are trivial. As reported in Section 5, the majority of the subjects (20 out of 35) used in the original study [11] can be fully covered in few seconds. We have run random search (RS) on the 33 subjects of the original study. In particular, we set RS with the same stop conditions used in LIPS and MOSA: either 100% of branch coverage is reached or the maximum budget of one minute is consumed. Results show that RS achieves 100% coverage in 18 out of 33 subjects (54%). It is also statistically equivalent to LIPS in other 9 subjects in terms of branch coverage. Thus, the large majority of subjects ($27/35 \approx 77\%$) used by Scalabrino et al. [11] are too easy to cover to draw any conclusion about the different performance of the two approaches. For this reason, we have extended the benchmark by adding more complex subjects.

5.2 Threats to validity

Threats to construct validity. Since the tool OCELOT is not publicly available, we had to re-implement LIPS in EvoSuite. While our re-implementation may slightly differ from the original one, we strictly followed the descriptions provided by Scalabrino et al. [11] with particular attention to the key contributions of LIPS (target selection order and dynamic budget allocation). As discussed in Section 5.1, our re-implementation is superior to the original one on the benchmark programs of the original study. Another construct validity threat regards the conversion of C functions to Java static methods. As indicated in Section 4, this conversion was straightforward since the considered functions do not have complex input parameters and do not involve advanced C constructs.

Threats to internal validity. Compared to the original study, we have increased the number of repetitions for MOSA and LIPS from 30 to 50 runs, to increase the statistical power of the analysis. We used the same termination criterion for LIPS and MOSA in terms of execution time. We used a different metric to measure efficiency, defined so as to remove the arbitrary constraint that both algorithms should consume 200,000 fitness evaluations.

Threats to external validity. To address the external validity threat of the original study, we have increased the size of the benchmark from 35 to 70 subjects by including methods with a larger number of branches (up to 425).

6 Conclusions and future work

We have replicated an empirical study comparing the test generators LIPS and MOSA. The former is a single-target approach with dynamic allocation of the search budget to each uncovered branch. The latter is a many-objective approach that targets all the branches at once. Our replication addresses several threats to external and internal validity of the preliminary study [11]. The new results differ remarkably from the original study: (i) MOSA is more effective than LIPS, especially for subjects with a large number of branches; (ii) MOSA is more efficient than LIPS, in terms of time needed to achieve the same, final coverage. Our implementation of LIPS together with the implementation of MOSA and a complete replication package are publicly available on GitHub as a fork of EvoSuite at the following link `https://github.com/apanichella/evosuite/tree/LIPS_replication`.

Our future agenda includes extending this study to non-procedural Java code and considering DynaMOSA [9], a recent, more advanced version of MOSA.

References

1. Baker, R.D.: Modern permutation test software. In: Edgington, E. (ed.) Randomization Tests. Marcel Decker (1995)
2. Conover, W.J.: Practical Nonparametric Statistics. Wiley, 3rd edition edn. (1998)
3. Deb, K., Deb, D.: Analysing mutation schemes for real-parameter genetic algorithms. Int. J. Artif. Intell. Soft Comput. 4(1), 1–28 (2014)
4. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast elitist multi-objective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comp 6, 182–197 (2000)
5. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Software Eng. 39(2), 276–291 (2013)
6. Fraser, G., Arcuri, A.: A large-scale evaluation of automated unit test generation using EvoSuite. ACM Trans. Softw. Eng. Methodol. 24(2), 8:1–8:42 (Dec 2014), `http://doi.acm.org/10.1145/2685612`
7. Juzgado, N.J., Vegas, S.: The role of non-exact replications in software engineering experiments. Empirical Software Engineering 16(3), 295–324 (2011)
8. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. 14(2), 105–156 (2004)
9. Panichella, A., Kifetew, F., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering PP(99), 1–1 (2017), pre-print available online
10. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST. pp. 1–10 (2015)
11. Scalabrino, S., Grano, G., Di Nucci, D., Oliveto, R., De Lucia, A.: Search-Based Testing of Procedural Programs: Iterative Single-Target or Multi-target Approach?, pp. 64–79. Springer International Publishing, Cham (2016)
12. Shull, F., Basili, V.R., Carver, J., Maldonado, J.C., Travassos, G.H., Mendonça, M.G., Fabbri, S.C.P.F.: Replicating software engineering experiments: Addressing the tacit knowledge problem. In: 2002 International Symposium on Empirical Software Engineering (ISESE 2002), 3-4 October 2002, Nara, Japan. pp. 7–16 (2002)
13. Shull, F., Carver, J.C., Vegas, S., Juzgado, N.J.: The role of replications in empirical software engineering. Empirical Software Engineering 13(2), 211–218 (2008)
14. Tonella, P.: Evolutionary testing of classes. In: ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 119–128. ISSTA '04, ACM (2004)
15. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)