The Seed is Strong: Seeding Strategies in Search-Based Software Testing

Gordon Fraser
Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Andrea Arcuri
Certus Software V&V Center at Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

***Abstract*—Search-based techniques have been shown useful for the task of generating tests, for example in the case of object-oriented software. But, as for any meta-heuristic search, the efficiency is heavily dependent on many different factors; *seeding* is one such factor that may strongly influence this efficiency. In this paper, we evaluate new and typical strategies to seed the initial population as well as to seed values introduced during the search when generating tests for object-oriented code. We report the results of a large empirical analysis carried out on 20 Java projects (for a total of 1,752 public classes). Our experiments show with strong statistical confidence that, even for a testing tool that is already able to achieve high coverage, the use of appropriate seeding strategies can further improve performance.**

***Keywords*-test case generation; search-based testing; testing classes; search-based software engineering**

I. Introduction

Search-based techniques have been shown to be a promising approach to tackle many kinds of software engineering tasks [1], particularly software testing [2]. Although automated generation of test cases for structural coverage has received particular attention, for example in the case of object-oriented software (e.g., [3]), such testing techniques are still not widely adopted by practitioners. This is partially due to current limitations in these techniques (e.g., in terms of efficiency and applicability), and because many of the different parameters that influence search-based software testing (SBST) are not well understood. Investigating these techniques is therefore of practical value.

In this paper, we consider the aspect of the initial population of the search, and present and study a series of techniques to improve search-based test data generation for object-oriented software. The objective of such a search is to automatically generate test suites that maximize branch coverage, while at the same time having the secondary objective of being as small as possible. This is necessary as many types of faults do not lead to crashes (e.g., segmentation faults and null pointer exceptions), and so the outcome of the test cases has to be manually verified (e.g., by controlling that the generated assert statements are capturing the intended behavior of the software). In this case, the test suites need to be small enough to be controlled by software engineers in feasible time. However, it is not clear what influence the seeding of the initial population has on the achievable results, and what are the best seeding strategies.

In our empirical analyses on this matter, we focus on the Java language, although the presented techniques can be extended to other languages as well. In particular, we study *seeding* strategies applied in three different contexts:

- Seeding of constants extracted from source code or bytecode (e.g., numbers and strings) throughout the search (e.g., initial population, mutation operators).
- Strategies to improve the initial population of the search in terms of diversity and suitability for the optimization target.
- Reuse of previous solutions (e.g., previously generated or hand crafted test cases) to seed the initial population of the search.

All these seeding techniques have been implemented in our automated testing tool EvoSuite [4]. EvoSuite is an advanced tool based on a Genetic Algorithm (GA), featuring for example the *whole test suite optimization* approach to test data generation [5], bloat control techniques [6], testability transformations [7] and effective assertion generation through mutation testing [8].

We evaluated these seeding techniques with a large case study, comprising 20 projects for a total of 1,752 public classes and 85,503 bytecode level branches, and we followed rigorous statistical procedures to assess the scientific validity of the presented results [9]. The results show that, with high statistical confidence, seeding strategies do improve the performance of the employed SBST technique. However, different strategies do provide different ranges of improvement, and in some cases this effect can be correlated with the type of the tested software (e.g., when the class under test makes strong use of string objects).

The paper is organized as follows: Section II sets up the context of seeding strategies in SBST, and Section III discusses different seeding strategies when testing object-oriented software. Section IV describes our experiments, and presents and interprets the results. Finally, we discuss threats to validity (Section V) and conclude the paper (Section VI).

II. Background

In this paper, with *seeding* we loosely refer to any technique that exploits previous related knowledge to help solve the testing problem at hand. The presence of this previous knowledge should not be a requirement to address the problem at hand (i.e., in theory the problem should be solvable even without using such knowledge).

The literature on evolutionary computation contains several papers on seeding strategies to improve the search. For example, in Genetic Programming, seeding strategies have been used in the context of improving different aspects of programs which the search should optimize. In the context of machine learning, Langdon and Nordin [10] for example studied a seeding strategy in order to improve the ability of a classifier/regressor to generalize. Similarly, White et al. [11] studied several different seeding strategies to initialize a Genetic Programming population for optimizing execution time of a given input program. Having each individual in the initial population being an exact copy of the input program could achieve that the search gets stuck in a sub-optimal area of the search landscape. Therefore, there is the need to use smart seeding strategies to re-use good "building blocks" from the original input program.

In the context of search-based software testing (SBST), the most common case of seeding regards the case when testing targets (e.g., branches to cover) are sought one at a time, as for example in [12]. The control dependence graph can be used to choose an order in which the targets are sought, and so reuse input data from previous runs when we seek to cover a dependent target. For example, if several targets are nested inside the same difficult branch, when we generate test data for one of them, we can re-use the result later as starting solution when we seek to cover the other targets, instead of re-starting the search from scratch (which can be expensive considering the difficulty of the parent branch in which they are nested).

In the context of testing real-time systems to find worst case execution times, Tlili et al. [13] applied seeding strategies as well. Given the execution time of the system under test as the fitness function to optimize, instead of starting from scratch, they used a test case with high coverage as seed to start the search from.

In some cases it can be useful to generate more test data, even if coverage for the chosen testing criterion is already maximized. This can be, for example, the case when automated oracles are available, or when software testers can afford to manually evaluate more test cases. Starting from a test case, Yoo and Harman [14] investigated a seeding strategy in which local search is applied on a solution test case, such that the diversity of the test data is maximized while the achieved coverage is not decreased. The rationale is that more different test cases would have more chances to find faults.

McMinn et al. [15] proposed seeding values taken from source code and documentation with the objective to reduce the human oracle costs. Similarly, Fraser and Zeller [16] used common object usage for seeding in the search to reduce the human oracle costs and to improve readability of the generated test cases.

A Genetic Algorithm (GA) usually starts from an initial population that is randomly generated. However, domain knowledge can be used to choose this initial population. For example, in test data generation there may be several targets that are not so difficult to cover. So it makes sense to do a first phase of random testing, before using a complex testing technique. For example, Miraz et al. [17] create the initial population by selecting the best individuals out of a larger pool of randomly generated individuals.

When testing software with predicates involving strings, generating the right strings for the input data can be very challenging, as the space of possible strings is much larger than, for example, the one of integers. Alshraideh and Bottaci [18] proposed and investigated a seeding technique in which the code of the SUT is analyzed, and then string constants are extracted and used as starting point for generation of string inputs. For example, consider the snippet `if(input.equals("complexAndLongString"))`: covering this branch becomes trivial, as the right input data would be present already in the first generation of the GA. As the seeded strings can be modified during the GA evolution (e.g., through the mutation operator), such seeding technique can be helpful even in more complex cases [18]. Besides extracting string constants from the SUT, a recent seeding approach has been investigated by McMinn et al. [19], in which candidate input strings are extracted from web queries on search engines based on SUT information.

Another recent seeding strategy in SBST has been proposed by Alshahwan and Harman [20], named "Dynamically Mined Value" seeding. In testing web applications, the resulting HTML pages generated as output of the test cases are then used as source of string inputs for the new test cases in the search.

III. Seeding Strategies for Class Testing

When generating tests for object-oriented code, the aim is to produce small sets of tests that maximize the coverage of the underlying code in the classes under test. In this section, we first describe the problem domain in more detail, and then show how different seeding strategies can be applied in this context.

A. Evolutionary Testing of Classes

A test suite for a class is a set of test cases, where each test case in turn is a sequence of statements (e.g., a simple JUnit test case). Each statement in a test case can generate objects through constructors and can access fields and methods. The length of test cases is typically variable, as is the number of test cases in a test suite, as it is highly dependent on the class at hand that is tested.

In our experiments, we use the EvoSuite test generation tool [4], which uses a GA to derive test suites for classes. Individuals of the population of the GA are test suites as described above. The GA works by iteratively selecting individuals from the population based on their fitness with respect to the search objective, and then applying crossover and mutation operators to the selected individuals. From generation to generation, the fitness of the individuals gradually improves, until either a solution has been found, or the search is terminated another way (e.g., when it hits a fixed bound on the number of generations or fitness evaluations). Crossover and mutation operators have to be defined specifically for each type of chromosome; in the case of test suite chromosomes, crossover amounts to exchange of test cases between two parent test suites, while mutation adds, deletes, or changes existing test cases. Changing a test case, in turn, involves deletion, change, and insertion of new method calls.

The search is guided by a fitness function that aims to maximize coverage [5]. For example, to measure the fitness of a test suite with respect to the common branch coverage criterion, we calculate the minimum branch distance [2] (estimate how close the branch was to being executed based on the guarding predicate) for each of the branches in the class under test, and essentially sum up the normalized branch distance values. An optimal solution thus has fitness 0.0. In practice, classes often have difficult branches that require the generation of complex sequences of method calls as well as specific constant values (e.g. numbers or strings). In the standard case, the initial population is generated randomly, and any constants generated during this initialization step or during the search are chosen randomly out of their respective value domains.

B. Seeding Constants

When branches are dependent on particular values, the program code often contains values that are similar to the sought values. For example, branch conditions often contain the boundary values as constants, and string comparisons are often performed with constant substrings, or matched against string patterns. An intuitive idea is to make use of such information by collecting constant values from the source code, and then seeding the search with these values. Even if the constant values in the code might not represent the exact values that are required to trigger certain branches, they are often close to those values.

The EvoSuite tool operates on bytecode, which is an intermediate representation used by many modern object-oriented languages: Compilation produces a binary representation of the classes that is close to machine code, yet retains parts of the original structure (e.g., classes and methods). Typically, bytecode also includes constant values and strings, and so it is easy to collect these constants from bytecode. In principle, constants can be extracted from source code just as well, if available.

This information is easily integrated into the search: During the search or initialization of the population, whenever attempting to generate a new constant value (e.g., to satisfy a parameter necessary for a method call), then with a certain probability P_{Bytecode} we use one of the collected constant values, rather than a random new value.

C. Optimizing the Initial Population

When the initial population is generated entirely randomly, it can be arbitrarily bad with respect to the optimization objective, even when it might be easy to produce a much better starting point for the search. Given some domain knowledge, it can be very easy to improve the initial population. In the case of testing classes, a prerequisite to achieve branch coverage is that every single method of the class under test is executed at least once. However, when generating random test cases there is no guarantee that all methods are called in the first place. Therefore, a simple strategy to improve the initial population is to attempt to maximize the number of different methods called in the initial population. For example, in EvoSuite we have implemented this strategy (which we call "AllMethods") such that during the initialization, each time a new method call is inserted, it is not chosen randomly; instead, it is chosen based on a ring buffer of all methods (i.e., each time a new method is requested, the next method in the buffer is returned). This does not guarantee that all methods are called by a single test suite; however, it is likely that all methods are called by some individual in the initial population.

Even when lacking domain knowledge, it is still possible to improve the initial population. For example, as discussed in Section II, $M > N$ test suites can be generated at random, and then based on some criteria N out of those M can be used as seed to initialize a GA population of size N. In particular, in this paper we consider the following strategy, which we call "Tournament": For each of the N positions we need to fill in the population, we generate 10 random test suites, and add only the one that has highest fitness. Notice that, compared to AllMethods, Tournament is more computational expensive, as it requires the evaluation of several test suites. To make a fair analysis, this computational cost is deduced from the search budget. In other words, a GA using Tournament would be run for fewer generations.

D. Incorporating Previous Solutions

When testing classes, often one does not start from scratch, but already has a certain set of test cases ready. These might for example originate from previous runs of the test generation tool, or they might be test cases written by hand by the developer of the class. Naturally, such information can be useful to seed the initial population.

In the context of testing classes, previous test cases are also sequences of method calls which need to be parsed and interpreted. This might be a non-trivial task, if a developer bases his tests on class hierarchies and interprocedural test calls. Given a set of parsed test cases T_P, the question is how to best use this information in the initial population. If T_P is small, then using this information too much might lead to the search being stuck in a sub-optimal area of the search landscape. On the other hand, if T_P is too large, then it may be difficult to map this to small individual test suites, as desired in evolutionary testing of classes.

When generating the initial population for the GA in EvoSuite, each initial test suite is given n random tests, where n is selected within predefined bounds. When given such a set of parsed test cases T_P, each time we produce a new test case, with probability P_{Clone} we do not produce it randomly, but clone an existing test case randomly chosen from T_P. Then, to increase diversity, we apply a number of mutations to this clone, chosen randomly in the range of $[0, N_{\text{Mutation}}]$.

IV. Evaluation

Having defined the different seeding strategies, we now address the following three research questions:

- RQ1: What is the impact of using constants from the bytecode for seeding?
- RQ2: Which are the best pre-processing techniques to seed an improved population before starting a SBST search?
- RQ3: Given a solution to improve from, which are the best seeding strategies to initialize a new population for SBST?

A. Case Study Subjects

For the evaluation, we chose a total of 19 open source libraries and programs. For example, among those there are several widely used libraries developed by Google and the Apache Software Foundation. Furthermore, to analyze in more details some specific types of software, we also used a translation of the String case study subjects employed by Alshraideh and Bottaci [18], and we also used a set of numerical applications from [21]. To avoid a bias caused by considering only open source code, we also selected a subset of an industrial case study project previously used by Arcuri et al. [22]. This results in a total of 3,165 classes, which were tested by only calling the API of the 1,752 public classes (the remaining classes are private and anonymous member classes).

[1]LOC stands for non-commenting lines of source code, calculated with JavaNCSS (http://javancss.codehaus.org/)

Table I
Number of classes, branches, and lines of code in the case study subjects

Case Study		#Classes		#Branches	LOC[1]
		Public	All		
COL	Colt	137	298	10,795	20,741
CCL	Commons CLI	14	15	662	1,078
CCD	Commons Codec	21	22	1,369	2,205
CCO	Commons Collections	246	421	8,683	19,190
CMA	Commons Math	247	306	10,503	23,881
CPR	Commons Primitives	210	231	2,874	7,008
GCO	Google Collections	85	370	4,214	9,886
ICS	Industrial Casestudy	21	29	373	809
JCO	Java Collections	30	118	3,531	6,339
JDO	JDom	57	61	4,098	6,452
JGR	JGraphT	137	193	2,467	5,924
JTI	Joda Time	131	199	8,681	18,003
NXM	NanoXML	1	1	310	661
NCS	Numerical Casestudy	11	11	209	421
REG	Java Regular Expressions	3	91	1,922	3,020
SCS	String Casestudy	12	12	607	606
TRO	GNU Trove	206	591	10,585	24,297
XEN	Xmlenc	7	7	1,645	788
XOM	XML Object Model	167	185	11,794	23,814
ZIP	Java ZIP Utils	3	4	219	441
Σ		1,752	3,165	85,503	175,564

The choice of a case study is of paramount importance for any empirical analysis in software engineering. To address this problem, in this paper we consider several types of software, as for example container classes, numerical applications, and software with high use of strings and arrays processing. To avoid bias in analyzing the results, we present and discuss the results of our empirical study grouped by project. In fact, different testing techniques can have comparatively different performance on different types of software. For example, random testing has been shown to be very effective in testing container classes [23]. If one only chooses container classes as case study and ignores for example numerical applications, then random testing could be misleadingly advantaged in technique comparisons. In fact, even if one uses several kinds of software in a case study, then it all depends on the proportion of the software types (e.g., if in the case study there are many more container classes than numerical applications). Therefore, aggregated statistics on all the artifacts of a case study need to be interpreted with care, as the proportion of different kinds of software types could lead to misleading results. Unfortunately, how to define a representative case study for test data generation is still an open research question.

B. Experiments

We carried out three different sets of experiments, each one addressing one of the three research questions. In all the experiments, we used the default settings of EvoSuite, where the search is stopped after either executing up to one million statements or a 10 minute timeout.

The choice of using the number of statements as stopping criterion is a common practice in the literature of SBST (and evolutionary computation in general), as it reduces the threats to internal validity regarding the implementation details of the developed research prototypes. This is reasonable only under the (usually met) assumption that the cost of the fitness function is "high" compared to the search operators (e.g., crossover and mutation). We also employed a 10 minute timeout for practical reasons. When using a large case study, there might be classes that are highly computational expensive, and this could be hard to detect without specialized tools before running the experiments. Due to the very large number of experiments on this case study, we used this 10 minutes timeout to be sure that all the experiments would be finished within a known time bound. Even so, the experiments took weeks to run even with the use of a cluster of computers.

All experiments were repeated 30 times to take the randomness of the employed algorithms into account, and the results were analyzed following the guidelines in [9]. When the algorithmic performance of two different algorithms/configurations is compared (e.g., which seeding strategy leads to higher branch coverage?), the effect sizes of the comparisons are quantified with the Vargha-Delaney $\hat{A}_{12}$ statistics. In our context, the $\hat{A}_{xy}$ is an estimation of the probability that, if we run EvoSuite with seeding configuration x, we will obtain better coverage than running it with configuration y. When two randomized algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{xy} = 1$ means that, in *all* of the 30 runs of EvoSuite with configuration x, we obtained coverage values higher than the ones obtained in *all* of the 30 runs with configuration y.

Statistical difference has been measured with a two-tailed Mann-Whitney U test. For reason of space, we cannot report all the p-values of the comparisons, so we only highlight the ones that are significant at $\alpha = 0.05$ level.

In the first set of experiments, we chose five different values for the probability P_{Bytecode} for the seeding from bytecode. In particular, $P_{\text{Bytecode}} \in \{0, 0.2, 0.4, 0.6, 0.8\}$, where $P_{\text{Bytecode}} = 0$ means no seeding at all (i.e., the default EvoSuite). Results of these experiments are presented in Table II. Notice that, for reasons of space, we only report the comparison of $P_{\text{Bytecode}} = 0.2$ with $P_{\text{Bytecode}} = 0$, as $P_{\text{Bytecode}} = 0.2$ was the configuration that gave best results on average on all the case study.

In the second set of experiments, we ran EvoSuite with the three different seeding strategies for the initialization of the first population. Results are presented in Table III. Notice that, to make the design of the experiments for each research question independent from each other, we used $P_{\text{Bytecode}} = 0$. Future work will be focused on studying the potential interaction effects among the different contexts for seeding strategies.

To gather empirical evidence for answering the last research question, we ran EvoSuite ($P_{\text{Bytecode}} = 0$ and default "Random" initialization strategy) on a subset of the case study for which hand-written test cases were available. Not all classes had hand-written test cases, and for some projects our parser was not able to handle their test cases. In particular, our parser currently does not handle inter-procedural test cases (e.g., when one test calls other functions), and it does not yet support Java Generics. This resulted in a total of four projects from which 156 public classes were used for the empirical study (roughly 8.9% of the entire case study).

Table II
For each project, average coverage on all of its classes when no bytecode constant is used ($P_{\text{Bytecode}} = 0$) and when they are used with probability $P_{\text{Bytecode}} = 0.2$. The $\hat{A}_{12}$ of these comparisons are calculated by aggregating all runs of all classes per project (in bold if statistically significant). On higher granularity, it is reported the percentage % of classes for which we have a significant $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.

Project	$P = 0$	$P = 0.2$	$\hat{A}_{12}$	$\% > 0.5$	$\% < 0.5$
COL	0.74	0.73	**0.48**	0.04	0.27
CCL	0.87	0.90	**0.56**	0.21	0.07
CCD	0.87	0.88	0.52	0.29	0.00
CCO	0.91	0.91	0.50	0.02	0.01
CMA	0.75	0.75	0.51	0.12	0.02
CPR	0.93	0.94	**0.52**	0.15	0.005
GCO	0.74	0.74	0.50	0.04	0.01
ICS	0.85	0.86	0.50	0.05	0.00
JCO	0.82	0.82	0.50	0.07	0.00
JDO	0.73	0.73	0.50	0.12	0.00
JGR	0.75	0.75	0.50	0.03	0.01
JTI	0.84	0.85	**0.52**	0.18	0.00
NXM	0.59	0.59	0.51	0.00	0.00
NCS	0.97	0.97	0.51	0.09	0.00
REG	0.75	0.75	0.50	0.00	0.00
SCS	0.63	0.85	**0.77**	0.75	0.00
TRO	0.88	0.87	**0.46**	0.005	0.32
XEN	0.65	0.72	**0.57**	0.29	0.00
XOM	0.76	0.77	0.51	0.17	0.00
ZIP	0.80	0.83	**0.69**	1.00	0.00
Average	0.79	0.81	0.53	0.18	0.04

We considered the combination of possible values for P_{Clone} and N_{Mutation}, in particular $P_{\text{Clone}} \in \{0.2, 0.4, 0.6, 0.8\}$ and $N_{\text{Mutation}} \in \{0, 4, 8, 16\}$. This resulted in 16 configurations for EvoSuite. We compared the performance of these configurations with the case of no seeding from hand-written test cases, which can be simply represented with $P_{\text{Clone}} = 0$. Results of the experiments for these configurations are presented in Table IV and Table V.

In Table IV, for each class and configuration, we calculated whether $\hat{A}_{12} > 0.5$ or $\hat{A}_{12} < 0.5$, and if such effect size is statistically significant at level $\alpha = 0.05$. Then, for each project, we calculated the difference $\mathcal{D}_{bw}$ among the number b of classes for which we have a significant $\hat{A}_{12} > 0.5$ and the number w of classes for which $\hat{A}_{12} < 0.5$, and normalize this difference based on the total number z of classes in that project used for the experiment, i.e. $\mathcal{D}_{bw} = (b - w)/z$. Finally, we average

Table III
FOR EACH PROJECT, AVERAGE COVERAGE ON ALL OF ITS CLASSES WHEN "ALLMETHODS" AND "TOURNAMENT" SEEDING STRATEGIES ARE EMPLOYED. THE $\hat{A}_{12}$ ARE IN RESPECT TO THE DEFAULT "RANDOM" STRATEGY, AND ARE CALCULATED BY AGGREGATING ALL RUNS OF ALL CLASSES PER PROJECT (IN BOLD IF STATISTICALLY SIGNIFICANT). ON HIGHER GRANULARITY, IT IS REPORTED THE PERCENTAGE % OF CLASSES FOR WHICH WE HAVE A SIGNIFICANT $\hat{A}_{12} > 0.5$ AND $\hat{A}_{12} < 0.5$.

Project	AllMethods				Tournament			
	Coverage	$\hat{A}_{12}$	$\% > 0.5$	$\% < 0.5$	Coverage	$\hat{A}_{12}$	$\% > 0.5$	$\% < 0.5$
COL	0.73	0.492	0.036	0.051	0.73	0.491	0.022	0.109
CCL	0.87	0.503	0.071	0.000	0.87	0.500	0.000	0.000
CCD	0.88	0.503	0.048	0.000	0.87	0.495	0.000	0.048
CCO	0.91	0.502	0.041	0.004	0.91	0.498	0.004	0.016
CMA	0.78	**0.521**	0.182	0.008	0.73	**0.482**	0.016	0.162
CPR	0.93	0.498	0.005	0.024	0.93	0.499	0.005	0.010
GCO	0.75	0.511	0.110	0.011	0.74	0.500	0.044	0.033
ICS	0.85	0.501	0.000	0.000	0.85	0.498	0.000	0.000
JCO	0.82	0.500	0.033	0.000	0.81	0.495	0.000	0.000
JDO	0.73	0.503	0.053	0.018	0.72	0.496	0.035	0.000
JGR	0.75	0.499	0.007	0.007	0.75	0.500	0.007	0.000
JTI	0.85	**0.513**	0.183	0.023	0.84	0.496	0.023	0.069
NXM	0.61	0.627	0.000	0.000	0.59	0.569	0.000	0.000
NCS	0.97	0.500	0.000	0.000	0.97	0.499	0.000	0.091
REG	0.75	0.501	0.333	0.333	0.74	0.487	0.000	0.000
SCS	0.63	0.501	0.000	0.000	0.64	0.504	0.000	0.000
TRO	0.88	0.500	0.010	0.010	0.88	0.499	0.010	0.019
XEN	0.65	0.502	0.000	0.000	0.65	0.499	0.000	0.000
XOM	0.76	0.499	0.000	0.024	0.76	0.499	0.000	0.012
ZIP	0.80	0.503	0.000	0.000	0.81	0.515	0.000	0.000
Average	0.83	0.509	0.056	0.026	0.82	0.501	0.008	0.028

Table IV
FOR EACH OF THE 16 COMBINATIONS OF P_{CLONE} AND N_{MUTATION}, THE TABLE REPORTS THE RESULTING $\mathcal{D}_{bw}$ AVERAGED FOR ALL THE EMPLOYED PROJECTS.

Clone Probability	Number of Mutations			
	0	4	8	16
0.2	0.15	0.13	0.15	0.15
0.4	0.14	0.17	0.17	0.19
0.6	0.18	0.17	0.19	0.17
0.8	0.18	0.20	0.18	0.18

these differences $\mathcal{D}_{bw}$ on all the four projects. Therefore, Table IV captures how often a seeding strategy configuration leads to get better results on a higher number of classes compared to the cases in which it brings to worse results. On the other hand, in Table V we show the details for the best configuration which can be derived from Table IV, i.e., $P_{\text{Clone}} = 0.8$ and $N_{\text{Mutation}} = 4$. Notice that $\mathcal{D}_{bw} = 0.20$ in Table IV is reflected in the difference $0.50 - 0.30$ in Table V.

C. Seeding Constants

The results in Table II show that the use of constants from the bytecode is beneficial for the search, with an average $\hat{A}_{12} = 0.53$, where the coverage is increased from 79% to 81%. At first look, this improvement might not appear significant, i.e., why bother about only a 2% increase? However, this magnitude is in line with expectations. Non-trivial software often have *infeasible* branches, and 100% coverage is impossible. Without a manual verification (which would not be possible due to the large number of classes involved), for what we can know 81% might be already the maximum (or close to) achievable average bytecode coverage. Some branches, even if feasible, could not be coverable (yet) by a tool such as EVOSUITE, because for example relying on the writing/reading of files (EVOSUITE currently has only limited support for files as test data). Furthermore, not all software rely on numerical constants and, even when they do, those might affect the control flow of only few branches.

Depending on how the case study is chosen, there can be a lot of variations in the results. This is why in this paper we employ a large and variegated case study, and why we present the results also grouped by project. For example, if we look at the details Table II we can see that the improvements for a project such as SCS are huge. Average coverage increases from 63% to 85%, with a statistically significant $\hat{A}_{12} = 0.77$. Improvements are statistically significant for 75% of the classes in SCS, and no case is present in which we get a class for which the results are statistically worse (this is in line with what was reported in [18], which is the reference from which the SCS project comes from). Therefore, the results in Table II show that, even on a large and variegated case study, seeding is beneficial, and there are cases in which it can be extremely beneficial. In our empirical study, we can for example see this phenomenon for projects that heavily rely on string manipulations.

At any rate, it is important to note that seeding, as for any heuristics, can be harmful in some cases. Although on average we obtained statistically better results for 18% of

Table V

DETAILS FOR THE BEST CONFIGURATION $P_{\text{CLONE}} = 0.8$ AND $N_{\text{MUTATION}} = 4$. THE $\hat{A}_{12}$ ARE IN RESPECT TO THE DEFAULT $P_{\text{CLONE}} = 0$, AND ARE CALCULATED BY AGGREGATING ALL RUNS OF ALL CLASSES PER PROJECT (IN BOLD IF STATISTICALLY SIGNIFICANT). ON HIGHER GRANULARITY, IT IS REPORTED THE PERCENTAGE % OF CLASSES FOR WHICH WE HAVE A SIGNIFICANT $\hat{A}_{12} > 0.5$ AND $\hat{A}_{12} < 0.5$. DETAILS OF THE CASE STUDY ARE PRESENTED AS WELL, AS THE NUMBER OF CLASSES, THE SIZE (MEASURED IN NUMBER OF STATEMENTS) OF THE HAND-WRITTEN TEST SUITES, AND THEIR COVERAGE.

Project	# of Classes	Coverage for $P_{Clone} = 0$	Seeding Coverage	$\hat{A}_{12}$	$\% > 0.5$	$\% < 0.5$	Seeded Test Suites Avg. Size	Seeded Test Suites Avg. Coverage
CCL	10	0.89	0.96	**0.64**	0.50	0.40	2032	0.77
CCD	15	0.92	0.94	**0.59**	0.47	0.20	52	0.67
JTI	69	0.90	0.92	**0.57**	0.54	0.35	776	0.65
XOM	62	0.71	0.82	**0.63**	0.48	0.24	2281	0.33
Average	39	0.86	0.91	0.61	0.50	0.30	1285	0.61

the classes, for 4% we obtained worse results. The cases in which we get worse results are concentrated in two projects: COL and TRO. However, in these projects the decrease in average coverage is only 1%. Therefore, in contrast to SCS, we have not seen any case in which seeding has drastic negative effects on performance. Understanding why we get worse results for COL and TRO will be important for future work to design better seeding strategies.

RQ1: Seeding constants from the bytecode improves performance, particularly for classes heavily relying on string objects.

D. Optimizing the Initial Population

In Table III we can see the results of experiments for the "AllMethods" and "Tournament" initialization strategies. In both cases, average coverage increases from 79% to 83% and to 82%, respectively. However, if we look at statistical effect sizes, those are not particular high, i.e., average $\hat{A}_{12} = 0.509$ for AllMethods, and $\hat{A}_{12} = 0.501$ for Tournament. When we look at the number of classes for which there is statistical difference, for Tournament it actually ends up that there are more cases in which this seeding strategy gives worse results (i.e., 2.8% compared to 0.8% of classes for which it gives better results). For AllMethods, we have statistically significant better results for 5.6% of classes compared to 2.6% in which we get worse results. If we consider projects as a whole, AllMethods gives statistically better results for two projects (CMA and JTI), whereas Tournament decreases performance for one (CMA).

The better performance of AllMethods, although small, can be explained as follows: When the class under test has only a few methods, the initial GA population will contain all of them with high probability, so AllMethods will have practical no effect on such type of classes. However, there are cases in which there are many methods in a class, as for example "LocalDateTime" in JTI that has 121 public methods. In EvoSuite, a random test suite used for the GA first population initialization would have $n \in [1,10]$ test cases, each one with length $l \in [1,50]$. On average, a test suite will have $(11/2) \times (51/2) = 140$ method calls chosen at random, with a maximum of $10 \times 50 = 500$. These two values (i.e., 140 and 500) might at first glance seem high compared to 121; however, counter-intuitively, the probability of containing at least one direct call to each of the 121 public methods is not high, which can be mathematically proven using the theory of the Coupon Collector's Problem [24]. On average, to cover all the 121 methods, we would roughly need $121 \times H_{121} = 650$ method calls (where H is the harmonic number), which is even higher than the maximum 500. By treating n and l as random variables, by using the theory of Coupon Collector's problem it would be possible to calculate the exact probability that a random test suite contains at least one call to each of the public methods. However, a detailed mathematical analysis of this process is not in the scope of this paper, and we refer the interested reader to [24] for further details.

RQ2: The AllMethods seeding strategy improves performance for classes with a high number of methods.

The not so particularly good performance of Tournament can have several related reasons, which would require a large amount of background material on evolutionary computation to be covered, which unfortunately we cannot include for reasons of space. First, Tournament is computational expensive, and it decreases the available budget for the GA search. The motivation behind Tournament is to have an initial population with fit individuals that are somehow diverse. However, when the search landscape has only few deceptive local optima (this does not mean that the search landscape is easy, as many plateaus could be present and monotonically connected toward the global optima, e.g., as in a ladder-like shape [25]), there would be no need to preserve/reward diversity in the population. A random individual that is fitter than the others will dominate the population in few generations, because is has higher chances to be selected for reproduction. The population will quickly converge to very similar and fit individuals, but that would not be a problem if there is no need to escape from deceptive local optima (i.e., in that case it is better to focus on the *exploitation* rather than the *exploration* of the search landscape).

Depending on details of the crossover operator, a premature convergence of the population to similar individuals would reduce its effects on the fitness of the offspring and the exploration of the search landscape. For example, crossing over two similar individuals might result in the same genotype with no visible effect on the phenotype, and so the search would be mainly driven by the mutation operator, which would put more focus on the exploitation of the search landscape. On the other hand, a crossover on two different but fit individuals would result in a very different genotype, whose phenotype quality would strongly depend on the effectiveness of the crossover operator to preserve and recombine building blocks.

EVOSUITE features the novel approach of evolving whole test suites at the same time, and there is still plenty of opportunities to improve search operators such as crossover. Therefore, future improvements on the crossover operator for test suite evolution would warrant the replication of this study on Tournament in the feature.

RQ2: The Tournament seeding strategy did not improve performance, although its performance might be strongly dependent on the crossover operator.

E. Incorporating Previous Solutions

The data in Table IV show that seeding from hand-written JUnit test cases is highly beneficial for any combination of P_{Clone} and N_{Mutation}. The data suggest that higher values for P_{Clone} are preferable, whereas N_{Mutation} does not show a clear trend.

The best configuration from Table IV is for $P_{\text{Clone}} = 0.8$ and $N_{\text{Mutation}} = 4$. Its performance details are presented in Table V. For all projects, there is strong statistical evidence that seeding provides better results (average $\hat{A}_{12} = 0.61$, and $\hat{A}_{12}$ values are statistically significant on each project).

However, it is important to notice that in this case, in contrast to the experiments that we ran for the other research questions, there is a human factor involved, i.e., the quality of the hand-written test cases. On one hand, if the test cases are poor, then even the best seeding strategy would likely have little impact on performance. On the other hand, if the hand-written test cases are optimal, then it would be pointless to try to improve upon them.

In our case study, the hand-written test cases have lower coverage (average 61%) compared to what can be obtained with EVOSUITE without any seeding (average coverage 86%). If we use seeding, then on average we obtained 91% coverage. This clearly shows the ability of SBST techniques to successfully exploit existing solutions and improve upon them.

RQ3: Seeding from existing hand-written test cases improves performance with high statistical confidence.

F. Search Budget

As discussed earlier, EVOSUITE is stopped after either executing up to one million statements or a 10 minutes timeout. However, the choice of this so called *search budget* can have large impact on the comparison of algorithms and their variants [26]. If the search budget is very “large”, likely (of course, exceptions exist) all the compared algorithms will solve the problem at hand with “high” probability (where what can be considered “large” and “high” is problem dependent). If the search budget is “low”, then none of the algorithms would solve the problem anyway, and so detecting differences could be difficult.

Ideally, one would compare search algorithms based on realistic values for the search budget, which could be derived from how practitioners actually use SBST tools. For example, a practitioner could run EVOSUITE while he is having a coffee break (e.g., five minutes), while at lunch (e.g., 40 minutes) or let it run on a Friday evening before leaving and collect the results on the next Monday morning (e.g., 66 hours).

One could suggest to run the algorithm only with a high search budget (e.g., 60 minutes), and then report performance at each predefined time interval (e.g. every five minutes). This would result in a graph of the performance (Y axis) over time (X axis). It would then be left to the readers and practitioners to decide which search budget applies to their testing contexts (e.g., 10 minutes rather than 30). Unfortunately, this argument would only apply on tools/techniques that are not tuned [26]. In fact, the performance of an algorithm is strongly correlated to its parameter settings, and the right tuning of parameters settings is correlated to the search budget [26]. For example, with a low search budget, one would use a smaller population size in GAs (to put more emphasis on the exploitation rather than the exploration of the search landscape) because, even if the search gets stuck in a local optimum, at any rate there would not be much budget left to explore other areas of the search landscape.

The choice of the search budget has an impact on the effect sizes we reported in the empirical analyses in this paper. As a clarifying example, in Figure 1 we show the performance of EVOSUITE at different search budget intervals for a specific class (all the other classes that we manually investigated resulted in similar graphs). In particular, we consider four different seeding strategies and no seeding. Figure 1 clearly shows that the differences in performance of the seeding strategies are more marked for low budgets, whereas they nearly disappear for higher search budgets.

The choice of search budget for the empirical analyses in this paper is based on our previous articles in which we used EVOSUITE, and it was rather arbitrary. However, at the current moment, it is hard to choose which representative budget to use, as usage statistics from practitioners are missing in the literature of SBST.

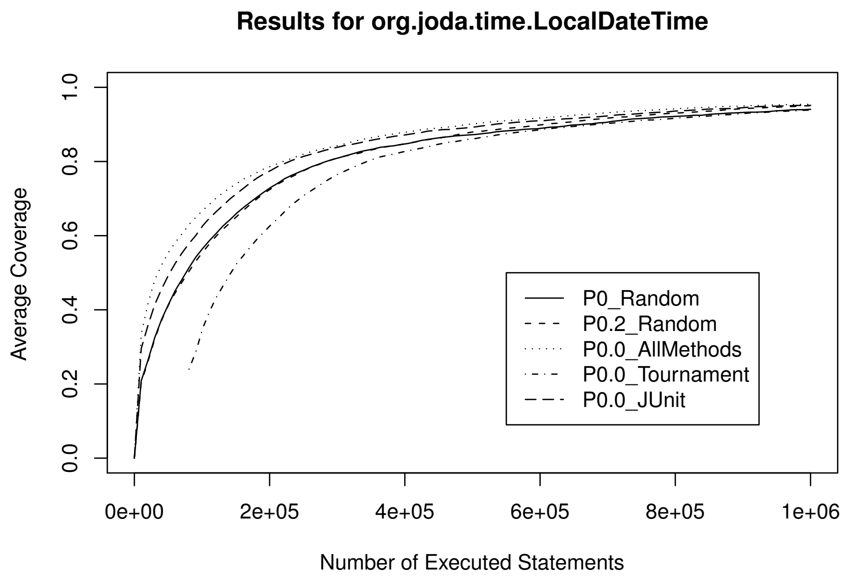

Figure 1. Coverage of the best individual during the search: Earlier during the search the difference between the seeding strategies is more significant, while at later points all strategies converge. P0_Random denotes random initial tests with no constant seeding, P0.2_Random uses constants with probability 0.2, etc.

V. Threats to Validity

Threats to *construct validity* are on how the performance of a testing technique is defined. We gave priority to the achieved coverage, with the secondary goal of minimizing the length. In this paper, for reasons of space, we have not considered the potential cases where, even if a seeding strategy leads to higher coverage, it might also lead to larger test suites. This yields two problems: (1) in practical contexts, we might not want a much larger test suite if the achieved coverage is only slightly higher, and (2) this performance measure does not take into account how difficult it will be to manually evaluate the test cases for writing assert statements (i.e., checking the correctness of the outputs).

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 30 times, and we followed rigorous statistical procedures to evaluate their results.

Another possible threat to internal validity is that we did not study the interactions/relations of the different parameter configurations of EvoSuite (e.g., population size and crossover probability) with the seeding strategies and the chosen search budget. In this paper we claim that seeding strategies help EvoSuite (and SBST in general) to achieve higher code coverage. However, in theory it might be possible that there exist parameter settings for which EvoSuite gives better results, and seeding might not improve upon them. To shed light on this possible issue, we would need to carry out large tuning phases and studying the possible correlations among all the different parameters (i.e., seeding strategies could be seen as further parameters to tune), but this would be very time consuming. At any rate, “default” parameter settings coming from the literature already tend to lead to reasonable performances [26].

Although we used a large case study consisting of 1,752 classes from both open source projects and industrial software, there is the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Furthermore, in this paper we evaluated the different seeding strategies only for EvoSuite. Therefore, the effectiveness of these seeding strategies might not generalize to other SBST techniques.

VI. Conclusions

Search-based testing techniques are dependent on a multitude of parameters and individual choices throughout the search. *Seeding* is one such technique that may strongly influence the result of an evolutionary search. In this paper, we analyzed the effects of different seeding techniques in the context of search-based testing for object-oriented languages. Our results provide evidence that a good choice of seeding techniques can lead to an overall improvement of the search results. In general, the more domain specific information can be included in the seeding strategies (e.g., previous solutions), the better the results will be. Assuming the search can find an optimal solution, the effects of seeding are larger during earlier phases of the search, while during later phases weaker seeding strategies may catch up.

In our experiments, we have considered several seeding strategies, and applied them to the context of testing object-oriented code in terms of the EvoSuite tool. Further seeding strategies are possible, and these as well as investigations of how individual seeding strategies interact will be part of our future work.

To learn more about EvoSuite, visit our Web site:

`http://www.evosuite.org`

Acknowledgments. This project has been funded by Deutsche Forschungsgemeinschaft (DFG), grant Ze509/5-1, and by a Google Focused Research Award on "Test Amplification". Andrea Arcuri is funded by the Norwegian Research Council. Thanks go to Yury Pavlov for his contributions to the JUnit parser, and Eva May for comments on an earlier version of this paper.

References

[1] M. Harman and B. F. Jones, "Search-based software engineering," *Journal of Information & Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

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

[3] P. Tonella, "Evolutionary testing of classes," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 119–128.

[4] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software." in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011.

[5] ——, "Evolutionary generation of whole test suites," in *International Conference On Quality Software (QSIC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 31–40.

[6] ——, "It is not the length that matters, it is how you control it," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 150 – 159.

[7] Y. Li and G. Fraser, "Bytecode testability transformation," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, M. Cohen and M. O'Cinneide, Eds. Springer Berlin / Heidelberg, 2011, vol. 6956, pp. 237–251.

[8] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 147–158.

[9] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.

[10] W. B. Langdon and P. Nordin, "Seeding genetic programming populations," in *Proceedings of the European Conference on Genetic Programming (EuroGP)*, 2000, pp. 304–315.

[11] D. White, A. Arcuri, and J. Clark, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation (TEC)*, vol. 15, no. 4, pp. 515–538, 2011.

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

[13] M. Tlili, S. Wappler, and H. Sthamer, "Improving evolutionary real-time testing," in *Genetic and Evolutionary Computation Conference (GECCO)*, 2006, pp. 1917–1924.

[14] S. Yoo and M. Harman, "Test data regeneration: Generating new test data from existing test data," *Software Testing, Verification and Reliability (STVR)*, 2010, http://dx.doi.org/10.1002/stvr.435.

[15] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation*, ser. STOV '10. New York, NY, USA: ACM, 2010, pp. 1–4.

[16] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *ICST 2011: Proceedings of the International Conference on Software Testing, Verification, and Validation*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 80–89.

[17] M. Miraz, P. Lanzi, and L. Baresi, "Improving evolutionary testing by means of efficiency enhancement techniques," in *IEEE Congress on Evolutionary Computation (CEC)*, 2010, pp. 1–8.

[18] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research articles," *Software Testing, Verification, and Reliability*, vol. 16, no. 3, pp. 175–203, 2006.

[19] P. McMinn, M. Shahbaz, and M. Stevenson., "Search-based test input generation for string data types using the results of web queries," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[20] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011, pp. 3–12.

[21] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[22] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *IFIP International Conference on Testing Software and Systems (ICTSS)*, 2010, pp. 95–110.

[23] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *Fundamental Approaches to Software Engineering (FASE)*, 2011.

[24] W. Feller, *An Introduction to Probability Theory and Its Applications, Vol. 1*, 3rd ed. Wiley, 1968.

[25] A. Arcuri, "Insight knowledge in search based software testing," in *Genetic and Evolutionary Computation Conference (GECCO)*, 2009, pp. 1649–1656.

[26] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering (SSBSE)*, 2011, pp. 33–47.