Bet and Run for Test Case Generation

Sebastian Müller, Thomas Vogel, and Lars Grunske

Software Engineering Group, Humboldt-Universität zu Berlin, Berlin, Germany
{muelerse, thomas.vogel, grunske}@informatik.hu-berlin.de

Abstract. Anyone working in the technology sector is probably familiar with the question: "Have you tried turning it off and on again?", as this is usually the default question asked by tech support. Similarly, it is known in search-based testing that metaheuristics might get trapped in a plateau during a search. As a human, one can look at the gradient of the fitness curve and decide to restart the search, so as to hopefully improve the results of the optimization with the next run. Trying to automate such a restart, it has to be programmatically decided whether the metaheuristic has encountered a plateau yet, which is an inherently difficult problem. To mitigate this problem in the context of theoretical search problems, the *Bet and Run* strategy was developed, where multiple algorithm instances are started concurrently, and after some time all but the single most promising instance in terms of fitness values are killed. In this paper, we adopt and evaluate the *Bet and Run* strategy for the problem of test case generation. Our work indicates that use of this restart strategy does *not* generally lead to gains in the quality metrics, when instantiated with the best parameters found in the literature.

Keywords: Search Based Testing · Test Case Generation · Bet and Run.

1 Introduction

Software testing plays an important role in providing evidence for the quality of software [17]. Due to the costs and complexity of writing tests manually, automating the generation of tests by metaheuristic search techniques is an active field of research [16,17,21]. A popular example for such a search-based approach to automated *test case generation (TCG)* is *EvoSuite* [5,6]. EvoSuite has shown its practical relevance by producing tests that achieve "good levels" of code coverage [7, p. 1] and detect real faults in open-source software [8]. The metaheuristic used by EvoSuite has evolved over time according to the state of the art, which is currently the *Dynamic Many-Objective Sorting Algorithm (DynaMOSA)* [18].

Due to the stochastic nature of metaheuristics, such algorithms are approximate and therefore "find in a reasonable computation time a solution that is as good as possible, but not necessarily optimal" [2, p. 242]. Exploring the search space stochastically, a metaheuristic typically yields different results with every run, and might get trapped in an inferior part of the search space (e.g., a plateau or local optimum) which leads to suboptimal results. As for any metaheuristic, this also applies to DynaMOSA in the context of TCG.

One way to improve the search results is to restart the metaheuristic whenever it is trapped in a plateau or local optimum to try out another run [12, 13]. However, such a restarting approach requires an appropriate strategy to be integrated in the metaheuristic that determines the trap and restarts the search [10]. To avoid such a metaheuristic-specific strategy and explicit restarts, Friedrich et al. [10] proposed *generic Bet and Run*, an approach that starts multiple short sample runs of a metaheuristic, evaluates the intermediate results of these runs after a certain point in time, and bets on the most promising run to continue the search until the search budget is used. This approach is generic as it is independent of the metaheuristic being used. This is in contrast to the original *Bet and Run* that is intertwined with the metaheuristic [4]. Thus, by starting multiple instances of a metaheuristic and selecting the most promising one, *Bet and Run* turns the problem of repeatedly restarting an instance on its head. First, there is no need to determine when a metaheuristic is trapped in an inferior part of the search space to trigger a restart. Second, the effect of repeated restarts is still obtained by starting and trying out multiple runs of the metaheuristic. The *generic Bet and Run* has been successfully evaluated on two theoretical problems (Traveling Salesperson and Minimum Vertex Cover) that are structurally different, so that Friedrich et al. [10] expect that *Bet and Run* is generally helpful.

In this paper, we adopt the idea of the *generic Bet and Run* strategy, transfer it to the TCG problem using EvoSuite/DynaMOSA, and evaluate its general feasibility for TCG and its effectiveness in comparison to the state of the art, being EvoSuite/DynaMOSA. In our *Bet and Run* approach, we split the total time budget for generating test cases for a subject into two phases. In the first phase, we start multiple instances of EvoSuite/DynaMOSA concurrently, each with the same configuration but a different seed. Since *Bet and Run* is generic, we can treat EvoSuite/DynaMOSA as a black box without having to change the metaheuristic. At the end of the first phase, we sample the intermediate results (test cases) in terms of the fitness score provided by EvoSuite. After the first phase, we bet on the most promising instance that continues generating test cases during the second phase, whereas all of the other instances will be terminated. Accordingly, variants of *Bet and Run* in terms of how many instances will be started concurrently, and how the total time budget is split into the two phases, are possible and will be investigated in this paper. EvoSuite as the state-of-the-art approach serves as our baseline. It starts a single instance of DynaMOSA to generate test cases for a subject over the total time budget. We evaluate the effectiveness—in terms of achieved code coverage—of our *Bet and Run* approach in a head-to-head comparison with this baseline, giving both approaches the same time budget to generate tests for 107 Java classes.

To the best of our knowledge, no previous study has applied a generic *Bet and Run* approach to a search-based software engineering problem, let alone TCG. Several TCG approaches in the literature, however, have considered restarts of search. The Alternative Variable Method (AVM), a variant of hill climbing, restarts the search with a randomly selected solution candidate to overcome local optima, that is, when the fitness cannot be improved (cf. [1, 9, 11]). Chan et

al. [3] use a restart (complete reset) as one way to forget test cases in adaptive random testing, which should reduce overheads of restricting the search space. Mathesen et al. [14] propose a metaheuristic whose global search proposes locations where the local search is restarted if a local minimum has been found. Finally, a genetic algorithm has been proposed that produces the offspring (test suites) randomly in a generation rather than by evolution if the current population lacks diversity, which has an effect of restarting the search [22]. All of these approaches provide metaheuristic-specific restart strategies that are intertwined with the metaheuristic. In contrast, *Bet and Run* lifts the restart strategy to a generic level that is independent of the metaheuristic being used.

The main goal of this paper is to evaluate the feasibility and effectiveness of a *generic Bet and Run* approach for the TCG problem. For this purpose, we investigate the following two research questions:

> **RQ1** Can *generic Bet and Run* be adapted to work on the TCG problem?
> **RQ2** Does *generic Bet and Run* show a significant improvement in the quality of the generated tests as measured by coverage metrics?

Accordingly, the primary contributions of this paper are:

(1) As the first study, we investigate the applicability of *generic Bet and Run* on the TCG problem using EvoSuite and DynaMOSA. This will answer RQ1.
(2) We conduct an empirical study to compare *Bet and Run* and default Evo-Suite/DynaMOSA on real-world 107 Java classes from the *SF110* corpus [7].
(3) We provide a statistical analysis of the effectiveness of *Bet and Run* and EvoSuite/DynaMOSA. This will answer RQ2.

2   Test Case Generation with EvoSuite and DynaMOSA

The context of our work is the automated *test case generation (TCG)* problem that is about generating good quality tests for a given software. Particularly, we focus on generating unit tests, for which we use *EvoSuite*. EvoSuite is a popular search-based "tool that automatically generates test cases with assertions for classes written in Java code." [5, p. 416]. The metaheuristic used in EvoSuite has evolved over time from whole test suite generation [6] to *DynaMOSA* [18].

The *Dynamic Many-Objective Sorting Algorithm (DynaMOSA)* [18] is the state-of-the-art many-objective genetic algorithm to solve the test case generation problem by redefining it into a many-objective problem. Conceptually, DynaMOSA works on each statement (i.e., target to cover) of the class under test individually, instead of trying to generate a test suite for all statements simultaneously. Thus, it breaks the complex task of generating a test suite for an entire class into more manageable smaller pieces. DynaMOSA also only computes test cases for targets that can be reached immediately: All branches that are still nested below other uncovered control flow nodes are temporarily ignored. Thereby, DynaMOSA reduces the number of targets that are to be covered simultaneously and thus decreases computational complexity.

Instance #

Fig. 1. Example: Behavior of *Bet and Run* with n instances and a given run time $t_{total} = t_f + n \cdot t_k$ with a maximum initial run time t_k for each instance. In this case instance #2 showed the most promise at t_k and was left running.

3 Bet and Run for Test Case Generation

Any metaheuristic can get stuck in local optima. Once trapped, such algorithms do not tend to break free of such a plateau for a while. Friedrich et al. [10] adopted and evaluated the approach *Bet and Run* for two general theoretical computer science problems. *Bet and Run* was initially presented by Fischetti and Monaci [4] for a sequential tree search method. Friedrich et al.'s approach is no longer metaheuristic-specific and makes use of the typical multi-processing architecture of modern hardware. The strategy is characterized by first starting a number of algorithm instances that are identical in terms of call parameters but contain different starting populations. After some time only the instance that showed the most promise in terms of fitness values is then kept running.

Currently in the field of TCG, no restart strategies are automatically applied. Unless the end-users intervene themselves, the optimization algorithm is simply run once, and once only. But since the typically used optimization algorithms in state-of-the-art tools such as EvoSuite are employing heuristics, the observations above hold true in this field as well. Therefore, *Bet and Run* strategies are generally applicable to the *Test Case Generation* problem as well.

In their 2018 paper [18], Panichella et al. state that DynaMOSA quickly increases test suite quality, once it starts to generate tests. Therefore, DynaMOSA is a good candidate for *Bet and Run*, as we can select the most promising instance early in the search process.

In our study, we adapt the general *Bet and Run* approach as follows:

Definition 1. Bet and Run adaptation to *TCG*: *Let t_{total} be the full time budget available to the optimization.*

(i) Starting phase. *First we start n instances of DynaMOSA simultaneously, for some fixed n. While all typical run parameters are held fixed, the seed*

for the random number generator used for the optimization is varied across all n instances.

(ii) Evaluation phase. *After the initially selected but fixed time t_k has passed, we evaluate all n instances for their fitness in terms of the overall fitness score computed by EvoSuite for the tests generated so far.*

(iii) Elitism phase. *We then kill the least fit $n-1$ instances. If there are two or more candidates with the exact same fitness score, we continue the first of those candidates.*

(iv) Run phase. *Let the single remaining (and so far most promising) instance continue until t_{total} is fully used up. That is, the remaining instance runs for $t_f := t_{total} - n \cdot t_k$.*

This means n, t_k, and t_{total} are free parameters of our adaptation of the general *Bet and Run* approach. An intuitive visualization is shown in Fig. 1.

Notation: Let p be the percentage so that $t_k := p \cdot t_{total}$. We then say $RESTARTS_p^n$ when we use n instances of the algorithm during the *starting phase* and $t_k = p \cdot t_{total}$ as the evaluation time.

4   Implementation

In this section we provide an overview of the TCG project EvoSuite [5], as well as the adaptations made by us, so that we may answer RQ1. We used EvoSuite from a self-compiled *1.0.7* snapshot from the *git* repository[1] This was done as the -generateMO API option to use the state-of-the-art DynaMOSA algorithm is not available yet in the official 1.0.6 release version of EvoSuite.

To answer RQ1, the first requirement is to make use of the EvoSuite framework so as to extend its functionality with the generic *Bet and Run* restart strategy. As this approach works on top of the tool itself, we do not need to change any code of EvoSuite directly. Instead, we may use it as a blackbox tool, that is, by using its command-line API we are able to get it to run tasks for us.

Therefore, it suffices to concentrate on providing a runner script implementing the *Bet and Run* strategy, along with a set of tools needed for saving and evaluating raw data. Apart from those scripts, all other program functionalities (i.e., genetic operators, metrics, problem encoding, etc.) are those implemented in the EvoSuite tool.

In most real-world tools, there is one limitation present: One cannot simply "pause" a given algorithm run at some arbitrary point during computation, evaluate the fitness at that point, and then decide whether to continue that run or not. Instead, many tools provide a timeout function, that allows the user to specify a runtime, after which the optimization is stopped and evaluated. For implementing the *Bet and Run* approach, one therefore needs to fully restart the most promising instance after the given evaluation. This, however, leads to two main strategies of coping with the tool limitation in terms of remaining runtime:

---

[1] Download of EvoSuite after commit https://github.com/EvoSuite/evosuite/commit/e26a6b725539370aa22976f041f3304972e201a1 on January 10, 2020.

- It is possible to give the remaining instance after the full restart a maximum runtime of $t_f + 1 \cdot t_k$, in order to emulate the theoretical idea of the generic *Bet and Run* approach, where "pausing" the initial n instances would be possible. However, this implies that *Bet and Run* has t_k more computation time, when compared to the normal algorithm run without the restart strategy.
- A second possibility is to give the most promising instance after the full restart a maximum runtime of t_f. In this case, the time limit ensures, that both the normal algorithm *and Bet and Run* do not take more than t_{total}.

In this paper, we decided to use the latter approach, so as to not give *Bet and Run* any more runtime than the baseline is given. This helps to make the results between both approaches comparable.

To run the prototype implementation of *Bet and Run*, EvoSuite as a standalone, blackbox tool only requires a runner script to build the command-line API calls. This paper provides two such runner scripts: the *evosuiteRunner*, and the *evosuiteBARRunner*. Both of those runners are automatically called by the main *experiments.py*. While the first runner simply provides an interface for running the standard EvoSuite implementation on many classes in parallel (i.e., the $RESTARTS^1_{100\%}$), the second one includes all changes necessary for the *Bet and Run* variants. Both runners also collect data provided by EvoSuite and write the raw values to a checkpoint file per class. Additionally, both scripts are able to parse this raw data and thus compress the verbose output to the metrics.

All used scripts, parameter and settings files, as well as the used EvoSuite sources are made available publicly with the online resources of this paper[2].

5   Evaluation

In this section, we present our experimental set up, lay out the methodology with which we analysed the raw data gathered, present and discuss the results of our analysis, and finally we briefly discuss threats to validity. We provide both of the interesting raw data sets with the online resources of this paper.

5.1   General experimental set-up

All experiments of this paper, and the algorithms they are based upon, are implemented in either *Java OpenJDK 1.8.0* or *Python 3.6.10*. Additionally, for building the EvoSuite files, a *Maven* installation is required. All other tool versions are listed in the online resources accompanying this paper.

Computational resources: The experiments were run concurrently on a *Dell R920* compute server with 4 *Intel Xeon E7-4880 v2* processors (60 cores) @ 2.5GHz, 1024GB RAM, and a *Suse Leap 15* OS (running *Linux* kernel 4.12.14). For both experimental approaches we used 40 parallel worker threads.

---

[2] https://www.doi.org/10.5281/zenodo.3903206

Baseline Selection: To investigate both the applicability (RQ1) as well as the effectiveness (RQ2) of the *Bet and Run* strategy, DynaMOSA is the natural baseline to compare with. Not only is it the current state-of-the-art algorithm in the field of *Test Case Generation*, it is also the algorithm extended in this study: We implemented the restart strategy using the EvoSuite tool (by use of its command-line API) and its implementation of the DynaMOSA algorithm.

Data Set Generation: Finding a set of classes that is both large enough to be representative of the state of the art in software development, as well as comparable to the studies in the field presents a certain challenge, as the adapted data sets are often not made easily available for the public.

The *SF100* corpus of classes is a collection of 100 Java projects from www.sourceforge.net. The data set was designed to be statistically representative. In 2014 Fraser et al. [7] accounted for the fact that *SourceForge* is home to a large number of older (and thus stale) projects, by revising the data set and adding the 10 most popular projects to it. This resulted in the data set now known as *SF110*[3]. This benchmark has been used in many studies in the field [19, 20], and most relevant to this study, it is also the main source of subjects for the study by Panichella et al. [18], in which they introduced DynaMOSA. Thus, to be comparable to their results, we chose to adopt the *SF110* benchmark. As we could not find the exact list of classes used by Panichella et al. in their 2018 study, we followed their class selection algorithm:

1. Compute McCabe's cyclomatic complexity [15] for each method in the SF110 dataset using the *CKJM* library[4].
2. Discard any project that only contains trivial methods (i.e., classes that only contain methods with a cyclomatic complexity of 1).
3. Randomly sample from the remaining projects.

Applying the above algorithm to the *SF110* corpus of classes resulted in a dataset of 107 different, non-trivial classes from 68 projects as provided by the *SF110* corpus of classes. The projects vary widely in size and difficulty: from a mere 18 branches on average (with 122 statements) in project `lavalamp`, to an average of 2515 branches in a medial 5887 statements in project `jmca`. This dataset was used for all experiments of this study. A complete list of all selected classes are made public with the online resources accompanying this paper.

5.2   Experimental set-up for RQ1

$RESTARTS^1_{100\%}$ is the standard run of a given algorithm without any restarts – and is thus ideally suited as a baseline. Friedrich et al. [10] have shown that $RESTARTS^{40}_{1\%}$ is the parameter set that yields the best results for both theoretical problems in their study. Of the non-trivial settings, we thus initially try $RESTARTS^{40}_{1\%}$, $RESTARTS^{20}_{2\%}$, and $RESTARTS^8_{5\%}$.

---

[3] http://www.evosuite.org/experimental-data/sf110/
[4] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

Therefore, to answer RQ1 and understand whether *Bet and Run* actually can work in the field of TCG, we ran an experiment using the above-mentioned computational settings, baseline, and dataset to test the four *Bet and Run* strategies.

The parameters for the *Bet and Run* experiments were thus set as follows: $n = 1, 40, 20$, and 8 initial runs, as well as $t_k = 100\%, 1\%, 2\%$, and 5% of t_{total}. The total time budget (t_{total}) was set to 5 minutes. In any run of the *Bet and Run* strategies t_{total} is split into $n \cdot t_k$ and t_f for the two phases (cf. Def. 1). Seeds for the initial state of the random number generator were selected at random for both the baseline and the initial phase of all employed *Bet and Run* strategy executions. For the *Bet and Run* experiments, we set the timeout t_k for all initial instance runs via the `-Dglobal_timeout` API parameter. On reaching time t_k, we evaluate each instance using the overall fitness score as provided by EvoSuite. This fitness score is an approach level (i.e., a distance) and means "lower is better". After that evaluation, we only run the single most promising candidate by restarting that instance with the timeout now set to t_f, in an effort to not give *Bet and Run* more runtime than the baseline. All four approaches were tested 10 times each in order to account for the pseudo-random nature of the employed DynaMOSA algorithm. A single experimental run of any given class took between approximately 60s to 5min. The entire experiment for answering RQ1 took about 3 full days per tested approach, when we used 40 concurrent worker threads.

5.3   Answering RQ1

In our first experiment, neither $RESTARTS_{1\%}^{40}$ nor $RESTARTS_{2\%}^{20}$ managed to generate any usable data. This is true for *all* 107 classes in the dataset. This problem is due to the fact that the requested test case generation time of the initial phase (t_k) – as required by the respective restart strategy – was too short for DynaMOSA to have managed to produce any tests while within the actual generation phase of EvoSuite. The restart strategy thus could not select any seed for the continued computation past t_k and therefore failed. In the $RESTARTS_{5\%}^{8}$ experiment, this particular problem happened to two of the selected classes: *Evaluation* of project *weka*, and *MethodWriter* of project *jiprof*. For all other classes $RESTARTS_{5\%}^{8}$ did manage to generate at least one test suite over the ten experimental runs and thus qualified for further evaluation.

> **Answer to RQ1:** The *Bet and Run* approach can indeed be adapted to yield usable results within the field of TCG, as shown by our experimental runs of $RESTARTS_{5\%}^{8}$.

5.4   Experimental set-up for RQ2

Using the result from RQ1, we select $RESTARTS_{5\%}^{8}$ for the statistical comparison to the baseline. To determine the effectiveness of the selected *Bet and*

Run strategy, we keep the dataset, computational set-up, and the parameters of both strategies the same as they were set to in the experiment for RQ1. We do, however, in order to generate a robust statistical evaluation repeat the execution of each strategy 30 times. Furthermore, we also log for evaluation the following fitness metrics upon fully using the time budget for each class.

Fitness Metrics: EvoSuite provides a number of different fitness metrics for each run by default: An overall fitness score, as well as Line, Branch, Exception, Weak Mutation, Output, Method, Method No Exception, and C Branch coverages. Additionally, we also track per class and per experimental run if EvoSuite encountered any internal errors during computation.

The overall fitness score is an aggregated *approach level* of the test suite to covering the remaining targets. As this approach level is a distance, it is to be read as "lower is better". Any coverage metric is a percentage of reached coverage and all are thus to be read as "higher is better." The according definitions used by EvoSuite may be found in [19], as well as in the source code of the tool. The number of internal EvoSuite errors, which is also tracked, is an absolute value, thus "lower is better".

The full experiment for RQ2 took another 18 days. Note, that these times do not include script run times for the subsequent evaluation of the raw data gathered.

5.5   Evaluation methodology for RQ2

The first step in answering RQ2, is to provide an overview over the actual behavior of the collected measures with regards to the two experiments. For *mean* values given in this paper the reader can safely assume "higher is better" for every metric, with the exception of the two measures *Time* and *Fitness Score.* The two excluded indicators need to be handled differently: The *Fitness Score* is an *approach level* of the test suite (cf. Section 5.4), where "lower is better". The measure *Time* has only informative character and shows how long each of the final instances spent in the actual generation phase.

In the subsequent statistical analysis (see Section 5.7), we employ the non-parametric *Wilcoxon Rank Sum* hypothesis test to determine statistical significance of our results for each class and each measure taken.

5.6   Answering RQ2

Generally, the results of our second experiment are structurally very similar across the different classes, as well as metrics logged. Raw data from both the *baseline* and the $RESTARTS_{5\%}^{8}$ for all classes (that generated at least one test case) and for all metrics are made available in the accompanying online resources. There we provide the *.zip of all raw data gathered, as well as a *.txt file with aggregated raw data, which was used in the evaluation below in this section.

We note that in the baseline experimental runs, 14 of the original 107 selected classes are rendered unusable for statistical evaluation, due to an excessively

high number of internal EvoSuite errors. Of the original set of classes in the $RESTARTS_{5\%}^{8}$ experiments, 15 are failures due to internal errors of the tool. Of all failures 12 classes fail too often for both approaches, 3 only when using DynaMOSA, and 4 only using *Bet and Run*. Of those 12 classes failing for both approaches 1 class completely fails to generate *any* test cases for either approach (class *Evaluation* of project *weka*). In total, this problem leads to 17 of the 107 classes not being eligible for statistical evaluation, apart from looking at which of the approaches was able to execute EvoSuite more stably.

5.7   Statistical Analysis and Discussion

In this section, we will present the results of the statistical analysis for the complete experiments of DynaMOSA and $RESTARTS_{5\%}^{8}$. We obtained the p-values using the non-parametric *Wilcoxon Rank Sum* hypothesis test, as provided by *SciPy*. Note that of the 107 initially selected classes only 90 were eligible for statistical evaluation due to internal EvoSuite errors. Table 1 shows per metrics collected the number of classes for which *Bet and Run* (BAR) is (i) identical to, (ii) worse than, and (iii) better than the baseline (BL). In each column, within the brackets we also show the number of statistically significant results of the total number of classes within that field. For instance, the first line is to be read as: For the metric *Fitness Score* there were a total of 28 classes that led to identical results in both approaches. Of those 28, 28 are statistically significant. Further there were 19 classes where the baseline outperformed the *Bet and Run* approach, of which 2 were statistically significant. Finally, there were 43 instances, where *Bet and Run* outperformed the baseline approach, of which only 4 classes were statistically significant.

   Table 1 shows that neither approach was significantly better than the other overall. In fact, in most cases the p-value either does not reach statistical significance or even indicates that both approaches reach statistically the exact same result. Over all 9 metrics and all 90 evaluated classes there were only 25 results that show any statistically significant *difference* between the two approaches. Of those, *Bet and Run* was 18 times significantly better than the baseline, and 7 times significantly worse (cf. Table 2). In contrast, over all 9 metrics and all 90 classes, we see 460 instances in which both approaches yield the exact same result. This means, when looking at *all* 810 results (i.e., 9 metrics x 90 classes) 785 instances are either only marginally different, or yield the exact same result.

   We argue that this is indicative of a limitation of the approach: In a real-world setting in the TCG context, runs typically are strictly limited in their computation times. This strict and more importantly *short* time budget for the genetic algorithm appears to mitigate the plateauing problem quite effectively. Furthermore, the use of the state-of-the-art DynaMOSA algorithm with its optimization mechanisms that work on minimizing the initial number of coverage targets (i.e., only search solutions for branches that can actually be covered at this time) help to further decrease the potential of getting stuck in a local optimum.

Table 1. Shown here are the aggregated results of the statistical analysis over all metrics when comparing the baseline DynaMOSA (BL) and the *Bet and Run* $RESTARTS_{5\%}^{8}$ strategy (BAR), where BAR = BL means both approaches show statistically the same result (p-value = 1.0), BAR < BL means the *Bet and Run* approach was worse than the baseline, and BAR > BL means *Bet and Run* was better than the Baseline. Any value inside of brackets indicate the number of statistically significant results in that field. Note that of the 107 initially selected classes only 90 were eligible for statistical evaluation due to internal EvoSuite errors.

Metric	BAR = BL	BAR < BL	BAR > BL
Fitness Score	28 (28)	19 (2)	43 (4)
Line Coverage	39 (39)	13 (1)	38 (3)
Branch Coverage	35 (35)	16 (2)	39 (4)
CBranch Coverage	36 (36)	16 (2)	38 (4)
Method Coverage	78 (78)	6 (0)	6 (0)
Method no Exception Coverage	69 (69)	11 (0)	10 (1)
Exception Coverage	80 (80)	5 (0)	5 (0)
Output Coverage	57 (57)	12 (0)	21 (1)
Weak Mutation Coverage	38 (38)	19 (0)	33 (1)

Panichella et al. state in their 2018 paper [18] that DynaMOSA quickly increases test suite quality, once it starts to generate tests. We can confirm this behavior: In our early experiments for RQ1, we saw that after a short initialization phase the algorithm managed to quickly improve fitness scores across all metrics for the population. We believe this quick improvement phase at the beginning is due to the fact that DynaMOSA only works on targets that can possibly be covered currently, instead of working immediately on every single target (i.e., the mechanism that the algorithm does not start working on targets that are still restricted by higher level, yet uncovered targets). This behavior is also a possible explanation for what we stated above: Neither $RESTARTS_{1\%}^{40}$, nor $RESTARTS_{2\%}^{20}$ managed to generate any tests. When we are using these settings, no candidate instance can be chosen to survive past the decision time t_k, as t_k was too short to get past the above-mentioned initialization phase. We note that while $RESTARTS_{1\%}^{40}$ was shown in Friedrich et al. [10] to be the best strategy, we cannot confirm this finding here. Moreover, we also cannot safely confirm quality increases in the test suite generated by EvoSuite. In fact, we would argue that the additional computational overhead is indeed an argument against usage of that particular restart strategy in the field of *Test Case Generation*, when also looking at the achieved results.

We have, however, seen a significant increase ($p = 0.002930$) in tool stability over all selected classes, when employing the *Bet and Run* strategy $RESTARTS_{5\%}^{8}$. While the *baseline* showed a total of 578 internal errors of the tool, the *Bet and Run* approach showed 500 errors across all classes. This difference of exactly 78 errors occurred over all 3210 experimental runs per approach (30 repetitions x 107 classes). That means in 15.6% of all experimental runs, the EvoSuite tool

Table 2. Shown here are the classes that reached a significant difference when comparing the baseline DynaMOSA and the *Bet and Run REST ARTS*$_{5\%}^8$ strategy.

Metric	Stat. sig. worse classes	Stat. sig. better classes
Fitness Score	99_newzgrabber: Downloader 12_dsachat: Handler	19_jmca: JMCAAnalyzer 61_noen: ProbeInformation 82_ipcalculator: BinaryCalculate 86_at-robots2-j: RobotRenderer
Line Coverage	12_dsachat: Handler	19_jmca: JMCAAnalyzer 82_ipcalculator: BinaryCalculate 86_at-robots2-j: RobotRenderer
Branch Coverage	39_diffi: IndexedString 12_dsachat: Handler	19_jmca: JMCAAnalyzer 61_noen: ProbeInformation 82_ipcalculator: BinaryCalculate 86_at-robots2-j: RobotRenderer
CBranch Coverage	39_diffi: IndexedString 12_dsachat: Handler	19_jmca: JMCAAnalyzer 61_noen: ProbeInformation 82_ipcalculator: BinaryCalculate 86_at-robots2-j: RobotRenderer
Method Coverage	-	-
Met. no Exc. Cov.	-	86_at-robots2-j: RobotRenderer
Exception Cov.	-	-
Output Coverage	-	50_biff: Scanner
Weak Mut. Cov.	-	61_noen: ProbeInformation

encountered errors during the $REST ARTS_{5\%}^8$ experiments, in comparison with 18.0% in the normal behavior EvoSuite experiments. This difference can be explained by the transience of internal EvoSuite errors, where a simple restart may fix the problem thanks to the selection of a new random seed value. As these transient internal tool errors usually happen very early during computation and since the *Bet and Run* restart strategy never selects any seed value that showed an error, this is expected behavior of the employed strategy. We assume the rather high number of internal errors by EvoSuite is due to the fact that we employed a self-compiled (and thus potentially unstable) tool snapshot, instead of an actual release of the software. We therefore argue, that unless tool stability is of particular importance to the use case, the statistically significant gains in this metric *do not* outweigh computational overhead needed for those increases.

> **Answer to RQ2:** Apart from statistically significant increases in tool stability, we cannot confirm any gains in the remaining nine metrics when using *Bet and Run* with the best parameters found in literature. In fact, we question the effectiveness of *Bet and Run* in the field of TCG, as time constraints and highly specialized algorithms (e.g., DynaMOSA) in this field effectively mitigate the typical plateauing problem of metaheuristics.

5.8   Threats to Validity and Future Work

In this paper, we rely on a search-based approach with a restart strategy to solve the Test Case Generation Problem. As such we see the following potential threats to the validity of our work.

Internal: In all genetic algorithms, a fair amount of (pseudo-)randomness is involved in the generation of their respective results. In DynaMOSA, the randomness problem is somewhat alleviated by the use of the archiving function – a test case covering some new target is never lost due to future mutations again. Additionally, we try to mitigate this threat by (i) sampling 30 runs per selected class and evaluating all results only in the average case; as well as (ii) providing online the seed values for all runs, both in the normal EvoSuite and the *Bet and Run* experiments (in the latter we provide the seeds that showed the most promise at time t_k). The seed values are made available in the aggregated result data files of the online resources. Furthermore, there exists the possibility of faults within our strategy implementation. We tried to minimize this threat by use of standard components (such as the EvoSuite implementation of DynaMOSA or the *SciPy* libraries for the statistical testing) wherever possible. All implementations, codes, and scripts are made available for inspection, review, and validation with the online resources accompanying this study.

External: Our evaluation is based upon the benchmark data set "*SF110 Corpus of Classes*" as provided by Fraser et al. [7]. While Fraser et al. argue that they selected 100 statistically representative *SourceForge* projects and even further enhanced this set with another 10 "popular" projects, the possibility that the heterogeneous nature of software is indeed not fully accounted for still exists: The restart strategy might behave differently for a different set of *real* classes. We leave for future work to revisit the use of the restart strategy with a different data set.

Construct: The first threat in this category that we can see is the possibility of the experiments not yet having encountered any plateaus. This threat could only be mitigated by redoing the entire set of experiments with a much higher timeout. However, this would go against real world usage of the EvoSuite tool. In practical settings, the tool is used to *quickly* generate a test suite for project classes – extremely high timeouts are thus the exact opposite of the intended use. We will, however, leave for future work a confirmation study with those increased timeouts: It is a matter of running the provided implementation with a new set of parameters. A second threat of this family is that while we selected restart strategy parameters that were shown to be "good" in the study by Friedrich et al. [10], "good" in the context of TCG might be something else entirely. In our case, it may be the case that most of t_k is in fact used for tool initialization, and no significant test generation is performed so as to correctly estimate the "best" performing one. Similarly to the first threat to construct validity, a parameter tuning study with a new set of parameters is left for future work, as

even more parameters for the restart strategy would go beyond the scope of this initial study. And thirdly, by forcing us to restart the most promising instance completely, the implementation limited us to choose between either giving the remaining instance in the run phase a maximum run-time of (i) $t_f + t_k$, or (ii) t_f. We chose the latter, so that both *Bet and Run* and the baseline do not use more than t_{total}. This leads to the evaluated instance having $p\%$ less run-time, while the total run-time of the *Bet and Run* approach is 100% of t_{total}. Had we used approach (i), however, *Bet and Run* would have had a total run-time of $t_{total} + t_k$.

6   Conclusion

In this study, we provided a tool to run and evaluate the generic restart strategy *Bet and Run* in the context of Test Case Generation using EvoSuite and the state-of-the-art DynaMOSA algorithm. To the best of our knowledge this was the first study that applies the generic *Bet and Run* approach to this field. Our work indicates that use of a restart strategy instantiated with the best parameters found in the literature does *not* generally lead to gains in the quality metrics: Not a single metric was improved by the *Bet and Run* restart strategy when compared with the EvoSuite/DynaMOSA as the baseline. In fact, for most selected classes, both approaches showed statistically the exact same results.

Only the stability of runs showed statistically significant improvements when *Bet and Run* was employed, as the number of internal EvoSuite errors decreased in the final generation process. However, from what we saw in our experiments, most of the internal EvoSuite errors are transient: In a real-world setting where the tool did not generate a test suite immediately, it is more than likely that a second (manually started) run with the same parameters would fix the problem.

Our results indicate, that (contrary to the promising results in the 2017 study by Friedrich et al. [10]) the restart strategy *Bet and Run* is *not* suited for improving the quality of automatically generated test suites using EvoSuite and its state-of-the-art DynaMOSA implementation considering the best *Bet and Run* parameters found in literature.

References

1. Baars, A., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.: Symbolic search-based testing. In: Proceedings of the 2011 26th International Conference on Automated Software Engineering. pp. 53–62. ASE '11, IEEE (2011)
2. Bianchi, L., Dorigo, M., Gambardella, L.M., Gutjahr, W.J.: A survey on meta-heuristics for stochastic combinatorial optimization. Natural Computing: An International Journal **8**(2), 239–287 (2009)
3. Chan, K.P., Chen, T.Y., Towey, D.: Forgetting test cases. In: Proceedings of the 30th International Computer Software and Applications Conference - Volume 01. pp. 485–494. COMPSAC '06, IEEE (2006)
4. Fischetti, M., Monaci, M.: Exploiting erraticism in search. Oper. Res. **62**(1), 114–122 (2014)

5. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: 19th Symposium on the Foundations of Software Engineering (FSE) and 13th European Software Engineering Conference (ESEC). pp. 416–419. ACM (2011)

6. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Software Eng. **39**(2), 276–291 (2013)

7. 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 (2014)

8. Fraser, G., Arcuri, A.: 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. Empirical Software Engineering **20**(3), 611–639 (2015)

9. Fraser, G., Arcuri, A., McMinn, P.: Test suite generation with memetic algorithms. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation. pp. 1437–1444. GECCO '13, ACM (2013)

10. Friedrich, T., Kötzing, T., Wagner, M.: A generic bet-and-run strategy for speeding up stochastic local search. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. pp. 801–807. AAAI Press (2017)

11. Kempka, J., McMinn, P., Sudholt, D.: Design and analysis of different alternating variable searches for search-based software testing. Theoretical Computer Science **605**, 1–20 (2015)

12. Lourenço, H.R., Martin, O.C., Stützle, T.: Iterated local search: Framework and applications. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics, pp. 363–397. Springer (2010)

13. Martí, R.: Multi-start methods. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, pp. 355–368. Springer (2003)

14. Mathesen, L., Yaghoubi, S., Pedrielli, G., Fainekos, G.: Falsification of cyber-physical systems with robustness uncertainty quantification through stochastic optimization with adaptive restart. In: 15th International Conference on Automation Science and Engineering, CASE. pp. 991–997. IEEE (2019)

15. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **2**(4), 308–320 (1976)

16. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2), 105–156 (2004)

17. Orso, A., Rothermel, G.: Software testing: A research travelogue (20002014). In: Future of Software Engineering Proceedings. pp. 117–132. FOSE 2014, ACM (2014)

18. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Trans. Software Eng. **44**(2), 122–158 (2018)

19. Rojas, J.M., Vivanti, M., Arcuri, A., Fraser, G.: A detailed investigation of the effectiveness of whole test suite generation. Empirical Software Engineering **22**(2), 852–893 (2017)

20. Shamshiri, S., Rojas, J.M., Fraser, G., McMinn, P.: Random or genetic algorithm search for object-oriented test suite generation? In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO. pp. 1367–1374. ACM (2015)

21. Tonella, P.: Evolutionary testing of classes. In: Avrunin, G.S., Rothermel, G. (eds.) Proceedings of the International Symposium on Software Testing and Analysis, ISSTA. pp. 119–128. ACM (2004)

22. Vogel, T., Tran, C., Grunske, L.: Does diversity improve the test suite generation for mobile applications? In: International Symposium on Search Based Software Engineering (SSBSE). pp. 58–74. Springer (2019)