Can RVEA with DynaMOSA features perform well at generating test cases?

Sergey Datskiv[1]

Supervisor(s): Annibale Panichella[1]**, Mitchell Olsthoorn**[1]**, Dimitri Stallenberg**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Sergey Datskiv
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Dimitri Stallenberg, Sicco Verwer

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Can RVEA with DynaMOSA features perform well at generating test cases?

Sergey Datskiv
Delft University of Technology
Delft, The Netherlands

ABSTRACT

On the intuitive level, software testing is important because it assures the quality of the software used by humans. However, ensuring this quality is not an easy task because as the complexity of the software increases, so do the efforts to test it. Search-based software testing is an active research field that develops and explores tools for automatic test case generation. Their work involves using meta-heuristic search optimisation approaches such as evolutionary algorithms from the evolutionary computation community and applying them to test case generation. The crossover between the evolutionary computation domain and the test case generation one produced DynaMOSA, a state-of-the-art evolutionary algorithm for generating test cases. In an attempt to produce another well-performing algorithm, this paper performs another crossover between the two communities and creates DynaMOSARVEA - a product of a Reference Vector Guided Evolutionary Algorithm (RVEA) and DynaMOSA. The conducted empirical study showed that although DynaMOSARVEA does not outperform DynaMOSA, it did outperform RVEA, thus demonstrating the value brought by domain-specific knowledge.

1 INTRODUCTION

According to Aniche most developers do not enjoy testing their software for various reasons, e.g. they claim that it is a time-consuming task which is not as fun as developing production code [3]. One way of testing the software is by writing unit-tests, which aim to ensure the functionality of the smallest part of the code, for example, a single method. However, it is still not an easy job because to test this method, developers need to understand its functionality, its application and isolate it if it has dependencies. This process is further complicated when the complexity of the software (e.g. the number of dependencies or parameters of a single method) increases.

Search-based software testing (SBST) is an active research field that aims at automating test case generation. The researchers from this field produced interesting tools which employ different approaches to test case generation, such as EvoSuite [17] and its evolutionary algorithm and Randoop [27] with its feedback-directed random testing. Recent studies have shown how these tools can achieve higher test coverage [2] and dramatically reduce the time needed for debugging and testing [32]. Furthermore, the SBST approach is also applied in the industry at companies like Meta, which developed its tool Sapienz [23].

DynaMOSA [29] is the current state-of-the-art approach to SBST implemented in many tools such as SynTest [25], Pynguin [21], EvoMaster [4] and helped the EvoSuite team to win in the last few SBST Tool Competitions [18]. DynaMOSA is an adaptation of a non-dominated sorting genetic algorithm II (NSGA-II [13]) from the evolutionary computation community to the test case generation domain. Using test case specific fitness functions, it evolves a set of initially randomly generated test cases over a given budget (e.g. time or number of generations). Software testing domain-specific knowledge makes DynaMOSA more effective at generating test cases than traditional evolutionary algorithms. [29]

Due to advantages such as the provision of many alternative solutions, wide applicability, low development costs, and other [14], evolutionary algorithms are an area of research on their own. The evolutionary computation community produced many algorithms apart from NSGA-II by Deb et al. Some algorithms try to improve various NSGA-II shortcomings, while others explore different approaches. Some of the popular work includes SPEA-II[1] [39], PESA-II[2] [10], RVEA [8], PCSEA[3] [35] and PSO[4] [33]. Although each algorithm is successful in its own right, they are not inherently designed for test case generation. Thus forms a knowledge gap because we do not know how those algorithms would perform at test case generation.

Panichella et al. picked NSGA-II as a base evolutionary algorithm and specialised it with domain-specific knowledge of test case generation, thus producing DynaMOSA. As studies [2, 6, 22, 30] and competitions [18, 31] showed it delivered impressive results at test case generation. Thus it is important to slowly start filling this knowledge gap to see if a different baseline algorithm adapted for test case generation problems could perform better than adopted NSGA-II (DynaMOSA) or if test case generation knowledge also improves other algorithms.

In this paper, we fill a part of a knowledge gap by investigating the performance of a Reference Vector Guided Evolutionary Algorithm (RVEA) in generating unit-test cases for JavaScript programs. Thus answering the following research questions:

(1) *How does RVEA adapted for test case generation perform at generating test cases automatically?*
(2) *Does domain-specific knowledge of test case generation problem makes RVEA perform better?*

RVEA is interesting because it relies on a decomposition-based approach that aims to produce Pareto-efficient solutions for problems, and instead of a non-dominance criterion, it combines convergence and diversity criterion for individual selection [8]. We chose to implement the algorithm to generate JavaScript test cases because of its popularity, as reported by [1, 37]. Specifically, we will work with the SynTest-Framework because it already implements the DynaMOSA-based test case generation for JavaScript programming languages.

[1] Strength Pareto Evolutionary Algorithm II (SPEA-II)
[2] Pareto Envelope-Based Selection Algorithm II (PESA-II)
[3] Pareto Corner Search Evolutionary Algorithm (PCSEA)
[4] Particle swarm optimization (PSO)

The conducted empirical study showed that the adapted version of RVEA for test case generation (DynaMOSARVEA) approaches the average branch coverage performance of DynaMOSA but misses it by one test file. But it (DynaMOSARVEA) outperforms RVEA at the average branch coverage in more than half of the test files, thus demonstrating the value brought by domain-specific knowledge.

The remainder of the paper is structured as follows; Section 2 provides necessary background knowledge, Section 3 explains our approach, Section 4 describes the study design, Section 5 presents the results, Section 6 discusses the threats to validity, Section 7 concludes the paper and suggests future work and Section 8 discusses responsible research.

2 BACKGROUND

2.1 SBST

Search-based software testing (SBST) uses various meta-heuristic search techniques to automate test case generation. The techniques include evolutionary algorithm, hill climbing, simulated annealing and tabu search [19]. Hill climbing and simulated annealing are local search approaches because they only one solution at a time, and only the neighbours near them as moving possibilities. In contrast, an evolutionary algorithm is a global approach because it considers many solutions at once [24].

As noted by McMinn, meta-heuristic techniques were successfully applied to problems including but not limited to functional, integration, and mutation testing. Furthermore, as mentioned in Section 1, the SBST approaches are successfully applied in the industry at companies such as Meta and many SBST tools, such as Sapienz, EvoSuite, and Randoop, was produced by academia and industry.

There are two requirements for applying a search-based approach to a problem. First, a possible solution to a problem has to be represented in code so an algorithm can manipulate it during the search process. E.g. in the case of a genetic algorithm, a subcategory of evolutionary algorithms, a solution has to be represented as an individual in a population which can be mutated and crossed over with other individuals. Second is the presence of a fitness function which will tell how close a given solution is to covering an objective. [24]

Depending on the testing task (e.g. branch coverage) and the selected approach (e.g. an evolutionary algorithm), the representation of the solution and choice of fitness function could differ. For example, when we apply a genetic algorithm to a branch coverage problem, we can model a solution in at least two ways.

One way is to represent a single solution (individual) as a whole suite, as proposed by Fraser and Arcuri. A single solution contains a variable number of test cases, while each test case has a variable number of method calls. In this case, a fitness function would consider how close the entire test suite (individual) is to covering all branches of a program [16]. This modelling of the problem is known as a single objective formulation since we have only one fitness function, which we will try to minimise.

However, Panichella et al. proposed a reformulation of the branch coverage problem as a many-objective problem (MaOP) if we consider each branch as an objective [28]. Therefore, we will have many objective functions, one for each branch, which has to be optimised (e.g. minimised). In this formulation, a solution (individual) is a single test case, not an entire test suite.

Several studies [2, 6, 22, 28–30] empirically showed the performance of the many-objective formulation of branch coverage problems at test case generation and how it outperforms the single objective formulation.

2.2 Overview of Evolutionary Algorithms

The evolutionary algorithm (EA) is motivated by Darwin's work on evolution and borrows concepts such as natural selection and survival of the fittest [15]. And it typically has four stages: initialisation, offspring creation, evaluation and selection.

The overall intuition is that by starting with a (random) population and a (fitness) function that tells us which individual is fitter than the other, we could select X[5] number of fittest individuals, exchange their genetic information and hopefully produce a fitter offspring population. The process loops from offspring creation to selection until it reaches a termination condition (e.g. exhausted budget, finding the best solution). Thus with each generation, the overall fitness of the population should increase. [15]

To apply an EA, a problem is represented in a way to allow the four stages to happen. E.g. a solution (individual) has to have genes (parameters) which could be exchanged with another individual through random variation operations like crossover and mutation. A problem-specific fitness function should be predefined to numerically identify individuals closer to achieving the desired objective. However, there could be more than one objective and fitness function.

In the evolutionary computation community, the algorithms and optimisation problems are classified based on the number of objectives (fitness functions). The categories are single, multi- and many-objective optimisation problems or evolutionary algorithms. Single refers to one fitness function or objective, multi- is between two and three, while many are more than three. The number of fitness functions also determines the dimensions of the objective space in which solutions are graphed. The need for this classification stems from the performance issues of the algorithms as the dimensions of the objective space increase.

2.3 Evolutionary Algorithms - Issues and Enhancement Approaches

The performance issues begin to appear when non-dominance is selected as a comparison measure between two different solutions in a population with a limited size [8]. According to the definition, a solution is non-dominated if there are no other solutions which are better than it in at least one objective and equally good or better in all other objectives [29].

For single-objective problems, the non-dominance comparison is usually not an issue because there is only one objective that is compared. A single-objective EA would find the best solution or several equally good solutions. However, MOEAs[6] and MaOEAs[7] have several objectives, which could be conflicting, meaning that

[5] X is a specified number of individuals, i.e. chosen population size.
[6] Multi-objective Evolutionary Algorithm (MOEA)
[7] Many-objective Evolutionary Algorithm (MaOEA)

one optimal solution does not exist instead there is a set of solutions, known as Pareto-optimal solutions, depicting the trade-offs between different objectives. The set of Pareto-optimal solutions forms a Pareto front (PF).

As Panichella et al. explain, the problem with Pareto-optimal solutions is that as the number of objectives increases, the size of a PF increases as well thus making it impossible to use non-dominance criteria to distinguish which solution is better than another. This problem is known as the loss of selection pressure.

Another performance issue of EA, highlighted by Cheng et al., stems from the difficulty of maintaining a good population diversity as the number of objectives increases. This is because the individuals in the population become more sparsely distributed as the number of fitness functions grows; creating more difficulties to the diversity managing strategies such as crowding distance of NSGA-II.

According to Cheng et al., there are roughly three categories of enhancements for evolutionary algorithms. First is convergence enhancement-based approaches which modify the dominance relationship, thus trying to increase the selection pressure toward the Pareto front. Second is decomposition-based approaches which break down a complex multi-objective problem into smaller multi-objective problems or into single-objective problems. Third is performance indicator-based approaches which are not subject to non-dominance relationship problems but rather high computational costs when the number of objectives is large.

2.4 RVEA

Cheng et al. proposed a Reference Vector Guided Evolutionary Algorithm (RVEA) for many-objective optimisation problems which is motivated by the ideas of the decomposition-based approaches. The algorithm is made of five parts, namely: reference vector generation, offspring creation, elitism selection, reference vector-guided selection and reference vector adaptation.

Reference vector generation either produces uniformly distributed unit vectors that decompose the original problem into several single-objective sub-problems or unit vectors targeting the user's preferred part of the Pareto front (PF).

The offspring creation happens through a random pairing of individuals to create a parent pair. The genetic material is exchanged through simulated binary crossover (SBX) and polynomial mutation operation.

Elitism selection is the result of offspring and parent populations combined into a single one before the start of reference vector-guided selection.

Reference vector-guided selection selects the fittest individual, according to the angle-penalised distance (APD) approach, in each subpopulation. Subpopulation is created by assigning each individual to a certain region created by the generated reference vectors. APD is a scalarisation approach designed to dynamically adjust two selection criteria, convergence and diversity, of the individuals as the search progresses. It picks the individuals closest to the ideal point[8] (convergence criterion) at the start of the search and individuals closest to the reference vector to which the individual belongs (diversity criterion) in the later stage of the search.

Reference vector adaptation adapts the reference vectors to the objective space current value range to mitigate the effects of objective values not falling within the same value range.

2.5 From NSGAII to DynaMOSA

The current state-of-the-art algorithm for test case generation, DynaMOSA, is a specialisation of a genetic algorithm NSGAII to the structural coverage problem[9]. NSGAII can be thought of as a baseline algorithm which was modified (specialised) with domain-specific knowledge of structural coverage problems to first create the MOSA algorithm, then the DynaMOSA.

The Many Objective Sorting Algorithm (MOSA) was introduced by Panichella et al. along with the many-objective problem formulation of the branch-coverage problem. In MOSA individuals are single test cases and their genes are the test input data and the possible program statements (e.g. method calls). MOSA objectives are the various structural coverage targets such as branch coverage, statement coverage and strong mutation coverage [29]. The fitness function[10] numerically scores how close the execution trace of an individual (test case) is to cover the desired targets (objectives). Therefore, the overall fitness of an individual is determined by how well its execution trace covers all objectives (e.g. branch coverage targets). Since the most interesting individuals for the structural coverage problem are the ones that cover the targets, Panichella et al. increased the selection pressure among the Pareto optimal solutions by introducing a preference criterion which prefers individuals closer to targets (objectives) and fewer program statements. Furthermore, Panichella et al. introduced an archive for storing the test cases that cover certain targets. The archive is updated with each generation if new objectives are covered or if an already covered objective has a new smaller test case. Lastly, the dominance comparator was changed to consider only uncovered targets instead of all as was originally introduced by Deb et al. in NSGAII [13].

DynaMOSA is MOSA which knows that there are structural dependencies among the targets (objectives) and uses this knowledge to dynamically select reachable objectives. To that end, DynaMOSA relies on a control dependency graph which shows the control dependency hierarchy. For intuition consider a program with nested if statements, the control dependency hierarchy would show that before reaching the inner statement, first, the outer one has to be satisfied. Therefore, DynaMOSA would first select the reachable outer statements, and once they are covered, it would consider the inner statements.

3 APPROACH

This paper presents our adaptation of RVEA to the test case generation problem and its modification with domain-specific knowledge. The result is three algorithms, RVEA, MOSARVEA and DynaMOSARVEA.

[8] In Cheng et al. paper which works with a minimisation problem, the ideal point is the origin of the coordinates.

[9] Structural coverage shows the amount of code exercised during a method call or a test.

[10] Each coverage criterion (e.g. branch coverage) has its own fitness function.

3.1 The RVEA Adaptation

Adaptation of RVEA to the test case generation problem required three adjustments namely, formulation of the structural coverage problems (e.g. branch coverage) as a many-objective one, the introduction of a population size parameter, and a slight modification of the search progress metric for angle-penalised distance (APD) calculation.

3.1.1 Many-Objective Formulation. As described in Section 2.2 to apply an evolutionary algorithm to a problem we first need to choose a representation for the individuals and their genes, as well as define the fitness functions. To that end, we adopted Panichella et al. many-objective formulation of the branch coverage problem as opposed to the single-objective formulation proposed by Fraser and Arcuri. The justification is that algorithms that adopted the many-objective formulation seemed to have performed better according to several studies [6, 22, 30]. This means that our individuals, objectives and fitness functions are the same as in MOSA. Individuals are single test cases and their genes are the test input data and the possible program statements (e.g. method calls). The objectives are the various structural coverage targets. The fitness functions are dependent on the chosen coverage criterion, e.g. the branch coverage is measured by the approach level and branch distance. Approach level refers to the amount of the control dependencies separating the objective and the execution trace of the individual [29]. Branch distance is the distance between the individual's predicate[11] value for a branch and the value that satisfies the branch predicate.

3.1.2 Population Size Dependent Reference Vector Generation. In the original RVEA algorithm population size itself is not a parameter but a byproduct of reference vector generation. Since reference vectors are responsible for decomposing the objective space into multiple subspaces they also set the upper limit on the population size at each generation because RVEA picks at most one individual from each subspace. If there are only two reference vectors, there will only be at most two selected individuals regardless of the starting population. To generate uniformly distributed reference vectors, Cheng et al. employed a method from [7]. This method has two parameters M for the number of objectives and H for the positive simplex-lattice design integer. First, it uses a canonical simplex-lattice design method [11]

$$\begin{cases} \mathbf{u_i} = (u_i^1, u_i^2, \ldots, u_i^M) \\ u_i^j \in \left\{\frac{0}{H}, \frac{1}{H}, \ldots, \frac{H}{H}\right\}, \sum_{j=1}^{M} u_i^j = 1 \end{cases} \tag{1}$$

to generate a set of uniformly distributed points on a unit hyperplane, where M is the number of objectives, H is a positive simplex-lattice design integer, and N in $i = 1, \ldots, N$ being the number of uniformly distributed points. Then, through normalisation

$$\mathbf{v_i} = \frac{\mathbf{u_i}}{||\mathbf{u_i}||} \tag{2}$$

of a vector $\mathbf{u_i}$ a corresponding unit vector $\mathbf{v_i}$ is obtained. The simplex-lattice design property allows us to calculate N, the total number of reference vectors (population size), through a combination formula $N = \binom{H+M-1}{M-1}$.

[11] Predicate is the conditional expression which creates the branching.

For the test case generation problem, the parameter M is always determined by the number of objectives derived from the unit under test. Cheng et al. follow the recommendations from [12] and [20] by deploying a two-layer vector generation strategy for problems where $M \geq 8$. However, their approach does not scale well for test case generation problems because the number of objectives can easily exceed eight and reach double-digit numbers and at times even triple-digits.

Instead of adapting a different approach to uniform reference vector generation, we decided to keep this methodology but adjust it to be based on the population size parameter (N), not H. For that, we first establish the minimum number of reference vectors that should be generated by choosing the highest value among a doubled number of objectives and the population size (`max(M*2, N)`). This condition ensures that if we have fewer objectives than the population size, then we will generate enough reference vectors to at least get the same population back (assuming that no subspace has more than one individual in it). If however there are more objectives, then this condition ensures that we will generate more than M vectors, i.e. more than one for each axis that creates the objective space. Second, we use $\binom{H+M-1}{M-1}$ to find the value of H which would create our minimum number of reference vectors. Once we found the appropriate H we generate reference vectors with equations 1 and 2 by passing the found H and our M.

3.1.3 APD Search Progress Modification. The original angle-penalised distance (APD) calculation relies on generation count to track the search progress. More concretely, the ratio of the current generation to the max generation ($\frac{t_{current}}{t_{max}}$) is used to indicate how far the progress has reached. However in test case generation, especially in a practical implementation like SynTest, there could be other ways to track the progress of the search (e.g. time budget). Hence in our algorithm, we track the search progress not necessarily through the generation count (passed iterations) but rather through the system's total progress ($\frac{Budget_{Current}}{Budget_{Total}}$).

3.1.4 Computational Complexity. As reported by Cheng et al., the computational complexity of RVEA is $O(M \times N^2)$ where M is the objective size and N is the population size. The above modifications do not increase or decrease the computational complexity since the test case generation reformulation takes place outside of the RVEA algorithm, reference vector generation still generates vectors according to the same method and the APD search progress value still performs the same calculation but acknowledges and includes other budget types.

3.2 MOSARVEA

To improve RVEA's performance at generating test cases we introduced the same domain-specific improvements as in MOSA, namely: preference criterion, archiving and focus on uncovered targets, thus creating MOSARVEA. The algorithm of MOSARVEA has a high resemblance with MOSA the main difference being the use of the RVEA algorithm in the PREFERENCE-SORTING subroutine of MOSA instead of a FAST-NONDOMINATED-SORT subroutine, as shown in Algorithm 1.

The Algorithm 1 is the PREFERENCE-SORTING subroutine from [29]. RVEAMOSA modifications are in blue, the rest is the same

as in the original subroutine. The subroutine begins with finding the zeroth front $\mathbb{F}_0$ using the preference criterion which picks the tests closest to the current uncovered objectives (lines 2-5). Then the algorithm removes the found tests from the total population (line 6) and checks if the population size was satisfied (line 7). If the population size is satisfied with $\mathbb{F}_0$ the rest of the test cases are assigned to $\mathbb{F}_1$ (line 8) and the algorithm returns. If the population size is not satisfied the algorithm proceeds to find more test cases using the RVEA's main subroutines (lines 10-18). The `else` condition starts with focusing the search only on the uncovered targets (line 10). Then, PREFERENCE-VECTOR-GENERATION generates reference vectors as described in Section 3.1.2. OBJECTIVE-VALUE-TRANSLATION contains objective values inside the first quadrant of the objective space and sets the coordinate origin to be the minimum value for each objective function (line 12), this is undone on line 15. POPULATION-PARTITION decomposes the problem into smaller single-objective sub-problems by assigning each individual to the closest reference vector, thus creating a sub-population. This is achieved by finding the maximum cosine value for each individual between the individual and all reference vectors (line 13), in the implementation a `map` is returned. Lastly, angle-penalised distance is calculated (line 14) and returns several fronts. Lines 16-18 append the found fronts to $\mathbb{F}$.

The key to substituting the FAST-NONDOMINATED-SORT subroutine with the RVEA subroutine was in adapting angle-penalised distance calculation to return multiple fronts. Usually, RVEA returns only the first front. However, we used the already calculated APD values to make the APD-CALCULATION return individuals who have higher APD scores as well. Choosing how to compose multiple RVEA fronts is something that can be investigated in future work.

Outside of the PREFERENCE-SORTING subroutine, we removed MOSA's calculation of crowding distance (see Appendix A) and RVEA's reference adaptation strategy. Crowding-distance is mainly used for ensuring the diversity of a population; however, RVEA's angle-penalised distance already accounts for the diversity. Although it would be interesting to see how crowding-distance would affect the algorithm, we decided to stick as much as possible to the original algorithm and leave experimentation for future work. The reference vector adaptation strategy, responsible for scaling the reference vectors to the values range of the objective space, was removed in preparation for dynamic target selection.

3.2.1 Computational Complexity. As reported in [29] the computational complexity of the FAST-NONDOMINATED-SORT subroutine is $O(M \times N^2)$ where M is the objective size and N is the population size. As was established in Section 3.1.4 the computational complexity of RVEA is also $O(M \times N^2)$. Therefore, the creation of MOSARVEA through the substitution of FAST-NONDOMINATED-SORT by RVEA should result in the same computational complexity as MOSA.

3.3 DynaMOSARVEA

Following in the steps of the original DynaMOSA we added dynamic objective selection to MOSARVEA, thus creating DynaMOSARVEA. Now MOSARVEA also uses the control dependency graph to derive structural dependencies among all of the objectives and consider the reachable statements first. The main side effect from adding dynamic target selection to RVEA is that the reference vector adaptation strategy is no longer needed because reference vectors are generated anew for each iteration of the algorithm because the dimensions of the objective space change.

3.3.1 Computational Complexity. DynaMOSARVEA should perform equally well or better than MOSARVEA by the proof from [29] because the modifications are the same.

Algorithm 1: PREFERENCE-SORTING

Input:
A set of candidate test cases T
Population size M
Result:
Angle-penalised distance (APD) ranking assignment $\mathbb{F}$

```
1  begin
2      F_0                          /* First non-dominated front */
3      for u_i ∈ U and u_i is uncovered do
           /* for each uncovered target we select the
              best test case according to the
              preference criterion */
4          t_best ← test case in T with minimum objective score
             for u_i
5          F_0 ← F_0 ⋃ {t_best}
6      T ← T − F_0
7      if |F_0| > M then
8          F_1 ← T
9      else
10         U* ← {g ∈ U : is uncovered}
               /* Perform RVEA-based front formation */
11         R_v ← REFERENCE-VECTOR-GENERATION(M,
             |U*|)
12         T* ← OBJECTIVE-VALUE-TRANSLATION(T,
             {u ∈ U*})
13         P ← POPULATION-PARTITION(T*, {u ∈ U*}, R_v)
14         E ← APD-CALCULATION(P, T*, {u ∈ U*}, R_v)
15         E ← UNDO-OBJECTIVE-VALUE-TRANSLATION(E,
             {u ∈ U*})            /* Pass E to undo the
             translation and update E */
16         d ← 0                  /* first front in E */
17         for All non-dominated fronts in E do
18             F_{d+1} ← E_d
```

4 STUDY DESIGN

4.1 Research Questions

To investigate if a different baseline algorithm adapted for the test case generation problem could perform better than the adopted NSGA-II (DynaMOSA) or if test case generation knowledge also improves other algorithms, we conduct an empirical study to answer the following two research questions (RQs).

RQ1 *How does RVEA adapted for test case generation perform at generating test cases automatically?*

RQ2 *Does domain-specific knowledge of structural coverage problem makes RVEA perform better?*

4.2 Benchmark

The benchmark we are using to test the performance of the algorithms was created in [36] by Stallenberg et al. It consists of five different projects, namely: Commander.js[12], Express[13], JavaScript Algorithms[14], Loadash[15] and Moment.js[16]. They were picked based on their popularity among the GitHub and Node Package Manager communities as well as their code style and syntax diversity.

Furthermore, only a subset of files for each project was picked. The files were selected based on their cyclomatic complexity (CC) to ensure that the CC is high enough so that high structural coverage is not achieved with few tests [36].

However, the Moment.js project was removed entirely from the benchmark because when running the experiment those files could not run. Application.js in express was removed because it started a web server which prevented the experiment from terminating.

4.3 Prototype

To conduct the experiment we made a fork[17] from SynTest-Core[18] repository and implemented RVEA, MOSARVEA and DynaMOSARVEA algorithms in TypeScript. SynTest-Core already has an implementation of DynaMOSA and the SynTest-Framework[19] provides the benchmark described in the benchmark section 4.2.

4.4 Parameter Settings

For this empirical evaluation, we chose not to experiment with different parameter settings but rather stick to the default values, since [5] showed that hyper-tuned parameters do not give a much better performance than the defaults. For DynaMOSA, DynaMOSARVEA and RVEA we used a single point crossover with a crossover probability of 0.7, and mutation with a probability of $\frac{1}{n}$ where n is the number of statements in the test case. Tournament selection was used for DynaMOSA but for RVEA and DynaMOSARVEA it is not a part of the algorithm. The population size of 50 was set for all algorithms. The search budget was set to 120 seconds instead of the common 60 seconds [26]. The additional minute gives the algorithms time to run a bit longer but not for too long. For RVEA and DynaMOSARVEA we set alpha, which controls the rate of change of the angle-penalised distance calculation, to equal two as in [8]. RVEA also required a parameter to control the frequency for the vector adaptation strategy, which was set to 0.1 like in [8].

4.5 Experimental Protocol

To answer RQ1 we compare the adapted version of RVEA (DynaMOSARVEA) with the DynaMOSA. The comparison is done using the average final branch coverage per file, unpaired Wilcoxon signed-rank test and Vargha-Delaney $\hat{A}_{12}$ statistic. It shows if a different baselines algorithm (RVEA) can be adapted to the test case generation problem and how it compares with the state-of-the-art DynaMOSA.

To answer RQ2 we compare RVEA[20] to DynaMOSARVEA. This comparison is done analogously to the comparison in RQ1 and shows how much domain-specific knowledge improves the performance of an algorithm.

Due to the algorithm's stochastic nature, we ran the experiment 10 times and calculated an average performance. Each run considers only one file from the benchmark and searches for the solution for 120 seconds, excluding the pre- and post-processing time. While the algorithm is taking 120 seconds to search for the solutions the number of covered objectives and timestamp is recorded for each generation of the search. Once the search exhausted its time budget, the final branch coverage and other statistics are recorded.

In total, we had 1080 runs because there were 36 benchmark files which were run for three configurations ten times ($36 \times 10\ times 3 = 1080$). Given the search budget of 120 seconds or 2 minutes, the run time of the benchmark would be $\frac{(1080 \times 2min)}{60min} \approx 36h$, excluding the pre- and post-processing time. We managed to run 100 instances in parallel, thus shortening the duration.

To prevent the hardware from making an impact on the experiment we used the same system, namely AMD EPYC 7H12 with a 64-Core Processor clocked at 3293.082 MHz with 512GB of RAM, 2 CPUs, 128 Cores and 256 Threads.

To compare the performance differences between the algorithms we calculated the average final branch coverage for each file. We chose to calculate the mean because in our preliminary runs[21] virtual in all cases the coverage per file per algorithm was the same.

We used the unpaired Wilcoxon signed-rank test [9] with a threshold of 0.05 (p-value ≤ 0.05) to determine if the experiments are significantly different from each other. This test checks if two data distributions can be sampled from the same one, if not they are significantly different from each other. Furthermore, to determine the magnitude of the difference between the two data distributions we paired the Wilcoxon signed-rank test with the Vargha-Delaney $\hat{A}_{12}$ statistic [38] to describe the effect size of the result.

5 RESULTS

This section presents the results, obtained by following the protocol explained in Section 4.5, for the two research questions from Section 4.1. Note that not all the benchmark files were included in the table because nine[22] of them could not generate any tests across all three algorithms. All of them were related to graphs and probably struggled with graph generation. All results can be found in the replication package[23].

[12] https://tj.github.io/commander.js/
[13] https://expressjs.com/
[14] https://github.com/trekhleb/javascript-algorithms
[15] https://lodash.com/
[16] https://momentjs.com/
[17] SynTest-Core Fork - https://github.com/SergeyDatskiv/syntest-core
[18] SynTest-Core - https://github.com/syntest-framework/syntest-core
[19] SynTest-Framework - https://github.com/syntest-framework

[20] Reformulated for test case generation problem but not enhanced with the knowledge of structural coverage problem
[21] Preliminary runs can be found in the replication package.
[22] The nine files are: articulationPoints.js, bellmanFord.js, bfTravellingSalesman.js, detectDirectedCycle.js, detectUndirectedCycle.js, eulerianPath.js, flyodWarshall.js, hamiltonianCycle.js, stronglyConnectedComponents.js.
[23] Replication Package - https://github.com/SergeyDatskiv/TUDelft-Bachelor-ResearchProject-CSE3000-ReplicationPackage

Table 1 can be split into four sections. First is the file under test provided by a benchmark project, second is the average branch coverage third and fourth are the p-value and $\hat{A}_{12}$ for two sets of comparisons. The average branch coverage shows how many branches each algorithm covered in each file, it is reported in percentage up to two decimal places. The highest-performing percentages are highlighted in grey, making the poorly performing algorithm easier to see. The next sections show p-value and $\hat{A}_{12}$ for RVEA vs DynaMOSARVEA comparison, with $\hat{A}_{12}$ values indicating the magnitude difference in brackets (Negligible, Small, Medium, Large). The baseline for statistical analysis in this case was RVEA and it was compared to DynaMOSARVEA. Analogously, the last sections show statistical analysis comparisons between DynaMOSA and DynaMOSARVEA. The baseline here was DynaMOSA and it was compared to DynaMOSARVEA.

Table 2 uses the p-value and $\hat{A}_{12}$ to classify the files under test into three categories a number of wins (# Win), no difference (# No Diff.) and loses (# Lose). To classify the results we first look at the p-value for a single test file for one comparison (e.g. DynaMOSA vs DynaMOSARVEA), if it is less than or equal to 0.05 (p-value $\leq$ 0.05) then it goes to either # Win or #Lose else it goes to # No Diff because the results are not significantly different. The classification between # Win and # Lose is based on the $\hat{A}_{12}$, if the $\hat{A}_{12}$ is less than 0.5 then it is a win for the statistical baseline, i.e. the algorithm on the left-hand side of the "vs". If it is bigger then, it is a lose for the left-hand side algorithm. The magnitudes (Negligible, Small, Medium, Large) are given by the function in `R` used to compute the $\hat{A}_{12}$ scores. As an example consider the DynaMOSA vs DynaMOSARVEA comparison, it has only one instance where p-value $\leq$ 0.05 which is in the response.js file. Since the $\hat{A}_{12}$ score is much less than 0.5 it is considered a win for DynaMOSA with the magnitude difference classified as large. All other files go to no difference column since their p-values are bigger than 0.5.

5.1 RQ1 Results

The results from Table 1 and 2 show that the adapted version of RVEA (DynaMOSARVEA) performs fairly well at generating test cases because its performance almost always matches the state-of-the-art algorithm (DynaMOSA).

The average branch coverage results from Table 1 shows that DynaMOSARVEA achieves the same coverage as DynaMOSA in 25 cases. The only flies where it does not achieve the same coverage are response.js and utils.js. However as can be seen by Table 2 only one of them, namely the response.js file, is significantly different to be categorised as a loss for DynaMOSARVEA (or a win for DynaMOSA). In all other cases, there is no statistically significant difference.

5.2 RQ2 Results

The results from Table 1 and 2 show that the domain-specific specialisations of RVEA to create DynaMOSARVEA have significantly improved the performance of the algorithm at generating test cases.

The average branch coverage results from Table 1 shows that RVEA achieves the same branch coverage as DynaMOSARVEA (or DynaMOSA) in 13 files out of 27. Table 2 shows that RVEA loses in total 15 times to DynaMOSARVEA meaning that the domain-specific improvements of DynaMOSARVEA allowed it to perform better in 15 cases. Thus quantifying the value which test generation knowledge brought to RVEA.

6 THREATS TO VALIDITY

Here we present potential threats to the validity of the conducted study.

Threats to internal validity. All algorithms which were compared in the results section are implemented on a fork of the SynTest-Core repository. We wrote a couple of tests to check if the subroutines of the algorithm perform correctly. However, tests alone do not guarantee that the code is bug-free. If bugs are present in any repository which was used to conduct the experiment, namely: SynTest-Core and its fork, SynTest-JavaScript and SynTest-JavaScript-Benchmark, then they could affect the results and validity. To mitigate the risk we did all of our work on a public fork of SynTest-Core and published all results and processing scripts. Thus, allowing others to inspect it and replicate the experiment or results.

Threats to external validity. The benchmark consists of five different projects featuring 36 test files so that the algorithms are tested on code of various purposes, sizes, syntax, application domains and code styles. However, the performance of the algorithm on those 36 test files is not the best generalisation approach. To increase the confidence in the generalisability of the study it would be beneficial to conduct a study with a bigger and more diverse benchmark. However, running such an experiment would incur a high cost. Hence, a smaller study is conducted here.

Threats to conclusion validity. To mitigate a possible positive or negative impact of a randomised nature of the algorithms, e.g. getting a good performance of algorithms due to an accidental good seed, each experiment was conducted 10 times with different random seeds. We used the Vargha-Delaney $\hat{A}_{12}$ effect size and unpaired Wilcoxon signed-rank test to quantify the significance and magnitude of the results.

7 CONCLUSION AND FUTURE WORK

In this paper, we reformulated RVEA for the structural coverage problem and specialised it with test case generation-specific knowledge. Thus producing three algorithms RVEA (for test case generation), MOSARVEA and DynaMOSARVEA. The empirical study measured the performance of the created algorithms by comparing them to the state-of-the-art algorithm for test case generation (DynaMOSA). We compared the performance of DynaMOSARVEA with DynaMOSA and found that DynaMOSARVEA has lower branch coverage only for one file. Furthermore, we compared RVEA with its improved version (DynaMOSARVA), thus quantifying the benefit of the domain-specific improvements which produced DynaMOSARVEA. Those improvements increased branch coverage in more than half of the tested files.

The time limitations and scope of the project prevented the paper from exploring more areas, many of which could be future work. The DynaMOSA and DynaMOSARVEA comparison can be analysed further by computing the area under the curve to see which algorithm reached the end branch coverage earlier. Moreover,

Table 1: Average branch coverage, p-value and $\hat{A}_{12}$ statistic for each benchmark file. Highest scores are highlighted in grey.

Benchmark	File Name	Average Branch Coverage			RVEA vs DynaMOSARVEA		DynaMOSA vs DynaMOSARVEA	
		RVEA	DynaMOSARVEA	DynaMOSA	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
Commander.js	help.js	25.76%	50.00%	50.00%	0.001	1 (Large)	0.1675	0.4 (Small)
	option.js	38.89%	50.00%	50.00%	0.01308	0.815 (Large)	0.3277	0.385 (Small)
	suggestSimilar.js	62.50%	71.88%	71.88%	0.00005	1 (Large)	0.1675	0.6 (Small)
Express	query.js	66.67%	66.67%	66.67%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	request.js	30.43%	32.61%	32.61%	0.0004	0.915 (Large)	1	0.5 (Negligible)
	response.js	10.87%	17.66%	19.57%	0.0001	1 (Large)	0.0007	0.055 (Large)
	utils.js	23.91%	41.30%	42.39%	0.00008	1 (Large)	0.6934	0.45 (Negligible)
	view.js	37.50%	37.50%	37.50%	0.1675	0.6 (Small)	0.5828	0.55 (Negligible)
JS Algorithms Graph	breadthFirstSearch.js	18.75%	18.75%	18.75%	1	0.5 (Negligible)	1	0.5 (Negligible)
	depthFirstSearch.js	0.00%	0.00%	0.00%	0.6809	0.55 (Negligible)	1	0.5 (Negligible)
	kruskal.js	20.00%	20.00%	20.00%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	prim.js	16.67%	16.67%	16.67%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
JS Algorithms Knapsack	Knapsack.js	57.50%	57.50%	57.50%	0.0344	0.7 (Medium)	NA	0.5 (Negligible)
	KnapsackItem.js	50.00%	50.00%	50.00%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
JS Algorithms Matrix	Matrix.js	6.58%	7.89%	7.89%	0.0146	0.75 (Large)	NA	0.5 (Negligible)
JS Algorithms Sort	CountingSort.js	57.14%	57.14%	57.14%	0.3681	0.45 (Negligible)	1	0.5 (Negligible)
JS Algorithms Tree	RedBlackTree.js	14.71%	29.41%	29.41%	0.00005	1 (Large)	NA	0.5 (Negligible)
Lodash	equalArrays.js	70.83%	83.33%	83.33%	0.00005	1 (Large)	0.3681	0.55 (Negligible)
	hasPath.js	100.00%	100.00%	100.00%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	random.js	85.71%	100.00%	100.00%	0.0001	0.955 (Large)	0.1681	0.4 (Small)
	result.js	80.00%	80.00%	80.00%	NA	0.5 (Negligible)	0.1675	0.4 (Small)
	slice.js	85.00%	100.00%	100.00%	0.00004	1 (Large)	NA	0.5 (Negligible)
	split.js	87.50%	87.50%	87.50%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	toNumber.js	55.00%	65.00%	65.00%	0.00002	1 (Large)	NA	0.5 (Negligible)
	transform.js	75.00%	91.67%	91.67%	0.0552	0.745 (Large)	1	0.505 (Negligible)
	truncate.js	44.12%	55.88%	55.88%	0.00002	1 (Large)	0.5828	0.45 (Negligible)
	unzip.js	100.00%	100.00%	100.00%	0.1681	0.6 (Small)	0.3681	0.55 (Negligible)

Table 2: Processed statistical results based on branch coverage.

Comparisons	# Win				# No Diff.	# Lose			
	Negl.	Small	Medium	Large	Negl.	Negl.	Small	Medium	Large
RVEA vs DynaMOSARVEA	-	-	-	-	12	-	-	1	14
DynaMOSA vs DynaMOSARVEA	-	-	-	1	26	-	-	-	-

since RVEA introduced reference vectors and hyper-parameters, it could be interesting to examine their impact on performance. E.g. investigating if a different reference vector generation strategy can lead to better performance. It could also be interesting to see if computing crowding distance or using tournament selection would improve DynaMOSARVEA's performance. Or if the non-dominated front can be computed using the convergence criterion of angle-penalised distance. Furthermore, more studies with a larger benchmark involving other algorithms (e.g. SPEA-II [39], PESA-II [10], PCSEA [35] and PSO [33]) could be conducted to examine to what extent the performance of DynaMOSA stems from NSGAII or the domain-specific improvements introduced by Panichella et al.

8 RESPONSIBLE RESEARCH

8.1 Ethics

Ethical concerns in our project could relate to the research stage and post-research state. The benchmark used to conduct our empirical study was created in [36]. In the Section 4.2, we explained how we used the benchmark, what we changed and why. Despite that, there could be ethical concerns in the post-research stage. The created algorithms aim to generate test cases for the developers to use, and because those algorithms are public, anyone could use them. Creating tests is not inherently wrong. However, the user has to be responsible for how they use the tool. Our algorithm does not make any promises regarding the quality of the tests, so the users of our algorithms must understand that we do not take responsibility

for the generated test cases. This line of thinking would be closest to the separatism notion and tripartite model explained by Poel and Royakkers in [34]. It claims that engineers can only be held accountable for the product design, not the consequences of its usage. When implementing the algorithms, we did not purposefully introduce any bias. Furthermore, we made our work publicly available for anyone to examine it and improve if there are problems.

8.2 Reproducibility

In academia, reproducibility is an important concept because it allows researchers to verify the claims of others. In our project, we relied on data from others, created our own and processed it. To ensure that our research is reproducible, we have a replication package[24] which contains the data generated during the project and the scripts for its extraction. Therefore, if someone would like to double-check our computations, for example, the statistical analysis, they will have access to the same data. Replicating the data itself is slightly trickier because of the randomness of the experiment and the algorithm. In our research, we tried to mitigate the stochastic nature of the empirical study by conducting several runs of our algorithms on the benchmark and reporting the average. The recorded data includes the settings of various parameters and the random seed. Therefore, others could try to produce the same results.

ACKNOWLEDGEMENTS

This paper is the last piece of my Bachelor's program at TU Delft. Therefore I want to take a moment to thank several people. First, a thank you to TU Delft for the education they provided throughout those three years. Second, a thank you to the research project staff for coordinating the project and assigning me to my research group. A big thank you to my responsible professor Annibale Panichella, and the supervisory team consisting of Mitchell Olsthoorn and Dimitri Stallenberg; without their guidance and support, this paper would not be possible. Another thank you to my research group peers who made working on this project more fun. Of course, a big thank you goes to all my friends who were with me through all these years. As much as I want to start naming you, I do not think I will finish the list, but thank you so much for everything. Last and most importantly, a big thank you to my entire family, who made all this possible and supported me at all times. I would not be where I am without all of you. I hope you are proud.

If I may dedicate this paper, then it would be to my Ukrainian aunt Nataliya and uncle Dmytro from Lviv, who is currently in Ukraine fighting for their freedom; may the peace come your way soon.

A APPENDIX A

This is the MOSA algorithm taken from [29]. It is included for completeness, the MOSA changes are in blue. The PREFERENCE-SORTING algorithm is 1

[24]Replication Package - https://github.com/SergeyDatskiv/TUDelft-Bachelor-ResearchProject-CSE3000-ReplicationPackage

Algorithm 2: MOSARVEA

Input:
$U = \{u_1, \ldots, u_m\}$ the set of coverage targets of a program.
Population size M
$G = \langle N, E, s \rangle$: control dependency graph of the program
$\phi : E \rightarrow U$: partial map between edges and targets
Result: A test suite T

begin
  U* ← targets in U with no control dependencies
  $t \leftarrow 0$ `/* current generation */`
  $P_t \leftarrow$ RANDOM-POPULATION(M)
  archive ← UPDATE-ARCHIVE($P_t, \emptyset$)
  U* ← UPDATE-TARGETS(U*, G, ϕ)
  while *not (search_budget_consumed)* **do**
    $Q_t \leftarrow$ RANDOM-POPULATION(M)
    archive ← UPDATE-ARCHIVE(Q_t, archive)
    U* ← UPDATE-TARGETS(U*, G, ϕ)
    $R_t \leftarrow P_t \bigcup Q_t$
    $\mathbb{F} \leftarrow$ PREFERENCE-SORTING(R_t, U*)
    $P_{t+1} \leftarrow \emptyset$
    $d \leftarrow 1$
    while $|P_{t+1}| + |\mathbb{F}_d| \leq M$ **do**
      ~~CROWDING-DISTANCE-ASSIGNMENT($\mathbb{F}_d$, U*)~~
      $P_{t+1} \leftarrow P_{t+1} \bigcup \mathbb{F}_d$
      $d \leftarrow d + 1$
    ~~Sort($\mathbb{F}_d$)~~ `/* according to the crowding distance */`
    $P_{t+1} \leftarrow P_{t+1} \bigcup \mathbb{F}_d[1 : (M - |P_{t+1}|)]$
    $t \leftarrow t + 1$
  T ← archive

REFERENCES

[1] [n. d.]. https://octoverse.github.com/2022/top-programming-languages
[2] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27
[3] Maurizio Aniche. 2022. *Effective Software Testing.* Simon and Schuster.
[4] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 394–397. https://doi.org/10.1109/ICST.2018.00046
[5] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (Feb 2013), 594–623. https://doi.org/10.1007/s10664-013-9249-9
[6] JosÃ© Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235. https://doi.org/10.1016/j.infsof.2018.08.010
[7] Ran Cheng, Yaochu Jin, Kaname Narukawa, and Bernhard Sendhoff. 2015. A Multiobjective Evolutionary Algorithm Using Gaussian Process-Based Inverse Modeling. *IEEE Transactions on Evolutionary Computation* 19, 6 (2015), 838–856. https://doi.org/10.1109/TEVC.2015.2395073
[8] Ran Cheng, Yaochu Jin, Markus Olhofer, and Bernhard Sendhoff. 2016. A Reference Vector Guided Evolutionary Algorithm for Many-Objective Optimization. *IEEE Transactions on Evolutionary Computation* 20, 5 (2016), 773–791. https://doi.org/10.1109/TEVC.2016.2519378
[9] W.J. Conover. 1999. *Practical Nonparametric Statistics.* Wiley. https://books.google.nl/books?id=n_39DwAAQBAJ

[10] David W. Corne, Nick R. Jerram, Joshua D. Knowles, and Martin J. Oates. 2001. PESA-II: Region-Based Selection in Evolutionary Multiobjective Optimization. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation* (San Francisco, California) *(GECCO'01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 283–290.

[11] John A Cornell. 1843. *Experiments with Mixtures*. Wiley-Interscience.

[12] Kalyanmoy Deb and Himanshu Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014), 577–601. https://doi.org/10.1109/TEVC.2013.2281535

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017

[14] A. Eiben and Jim Smith. 2003. *Introduction To Evolutionary Computing*. Vol. 45. https://doi.org/10.1007/978-3-662-05094-1

[15] D.B. Fogel. 2000. What is evolutionary computation? *IEEE Spectrum* 37, 2 (2000), 26–32. https://doi.org/10.1109/6.819926

[16] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[17] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (dec 2014), 42 pages. https://doi.org/10.1145/2685612

[18] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. 2022. SBST Tool Competition 2022. In *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. 25–32. https://doi.org/10.1145/3526072.3527538

[19] Manju Khari and Prabhat Kumar. 2017. An extensive evaluation of search-based software testing: a review. *Soft Computing* 23, 6 (Nov 2017), 1933–1946. https://doi.org/10.1007/s00500-017-2906-y

[20] Ke Li, Kalyanmoy Deb, Qingfu Zhang, and Sam Kwong. 2015. An Evolutionary Many-Objective Optimization Algorithm Based on Dominance and Decomposition. *IEEE Transactions on Evolutionary Computation* 19, 5 (2015), 694–716. https://doi.org/10.1109/TEVC.2014.2373386

[21] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/3510454.3516829

[22] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* 28, 2 (Jan 2023). https://doi.org/10.1007/s10664-022-10248-w

[23] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[24] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 153–163. https://doi.org/10.1109/ICSTW.2011.100

[25] Mitchell Olsthoorn, Dimitri Stallenberg, Arie van Deursen, and Annibale Panichella. 2022. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. In *The 44th International Conference on Software Engineering-Demonstration Track*. IEEE/ACM.

[26] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1224–1228.

[27] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[28] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102604

[29] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435

[30] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256. https://doi.org/10.1016/j.infsof.2018.08.009

[31] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. 2021. SBST Tool Competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. 20–27. https://doi.org/10.1109/SBST52555.2021.00011

[32] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. 2016. The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 547–558. https://doi.org/10.1145/2884781.2884847

[33] K. E. Parsopoulos and M. N. Vrahatis. 2002. Particle Swarm Optimization Method in Multiobjective Problems. In *Proceedings of the 2002 ACM Symposium on Applied Computing* (Madrid, Spain) *(SAC '02)*. Association for Computing Machinery, New York, NY, USA, 603–607. https://doi.org/10.1145/508791.508907

[34] Ibo van de Poel and Lambèr Royakkers. 2011. *Ethics, Technology, and Engineering*. John Wiley Sons.

[35] Hemant Kumar Singh, Amitay Isaacs, and Tapabrata Ray. 2011. A Pareto Corner Search Evolutionary Algorithm and Dimensionality Reduction in Many-Objective Optimization Problems. *IEEE Transactions on Evolutionary Computation* 15, 4 (2011), 539–556. https://doi.org/10.1109/TEVC.2010.2093579

[36] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2022. Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference. In *Search-Based Software Engineering*, Mike Papadakis and Silvia Regina Vergilio (Eds.). Springer International Publishing, Cham, 67–82.

[37] TIOBE. 2022. TIOBE Index | TIOBE - The Software Quality Company. https://www.tiobe.com/tiobe-index/

[38] András Vargha, Harold D. Delaney, and Andras Vargha. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101. https://doi.org/10.2307/1165329

[39] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001-05. *SPEA2: Improving the strength pareto evolutionary algorithm*. Report. Zurich. https://doi.org/10.3929/ethz-a-004284029