Investigating the performance of SPEA-II on automatic test case generation

Erwin Li[1]

Supervisors: 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: Erwin Li
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Dimitri Stallenberg, Sicco Verwer

ABSTRACT

Software testing is an important but time-consuming task, making automatic test case generation an appealing solution. The current state-of-the-art algorithm for test case generation is DynaMOSA, which is an improvement of NSGA-II that applies domain knowledge to make it more suitable for test case generation. Although these enhancements are applicable to other evolutionary algorithms, no research has been done on how effective other algorithms can function as the base. In this paper, we apply the DynaMOSA modifications to SPEA-II to create a new algorithm, DynaSPEA-II. We conduct an empirical experiment where we evaluate the DynaMOSA enhancements, and directly compare DynaSPEA-II to DynaMOSA. The algorithms are assessed on a benchmark consisting of 36 diverse JavaScript classes w.r.t. branch coverage. Our results show that adding DynaMOSA enhancements to SPEA-II results in higher coverage in 13.9% of classes, with an average increase of 4.92% for classes where a statistically significant difference was found. DynaSPEA-II performed equally to DynaMOSA, with no statistically significant difference being found between the two.

1 INTRODUCTION

Software testing is a crucial part of development to ensure the quality and reliability of programs. However, manually writings tests can be time-consuming and prone to errors. Moreover, with the increasing complexity of programs, designing effective test cases has become more and more difficult. Therefore, the development of automated test case generation has been an active research topic [10], and has managed to produce approaches that produce greater coverage than manually-written tests [1]. Furthermore, automatic test case generation has been successfully applied in industry [2], and has helped developers reduce time spent on testing and debugging [16].

This paper focuses on the use of genetic algorithms for automatically generating test cases, specifically in the context of Javascript programs. The current state-of-the-art tool for automatic test case generation in JavaScript is *Syntest-Javascript*[1], which implements the DynaMOSA algorithm. DynaMOSA is a many-objective solver that uses domain knowledge to make it more efficient in generating test cases [14]. DynaMOSA is an improved version of MOSA, which in turn extends NSGA-II, a widely used multi-objective genetic algorithm that has been successfully applied to many problems. Domain knowledge needs to be incorporated, as classical multi-objective evolutionary algorithms, like NSGA-II, are most effective in solving problems with up to three objectives [8]. However, in the context of test case generation as a many-objective problem, every branch is an individual objective, leading to hundreds, if not thousands of targets. Although the enhancements introduced by DynaMOSA are applicable to various evolutionary algorithms, to the best of our knowledge, there has been no research conducted to investigate whether employing a different base algorithm can lead to improved results.

The goal of this research is to investigate how well the improved Strength Pareto Evolutionary Algorithm (SPEA-II) performs in automatic test case generation [23]. SPEA-II is another algorithm belonging to the evolutionary family, with potential for promising results in automatic test case generation, as it has been successfully applied to solve multi-objective problems before. Furthermore, it has shown better performance than NSGA-II in some problems with higher-dimensional objective spaces [23]. Finally, it will be augmented with the core ingredients of DynaMOSA to create a novel algorithm DynaSPEA-II, to see whether this yields improved results.

The main research questions of the paper are: (1) *How well does the DynaSPEA-II algorithm perform in automatic test case generation?* (2) *How effective are the DynaMOSA enhancements to* SPEA-II *for automatic test case generation?* To answer these questions, we performed an empirical study on a dataset of diverse Javascript classes, sourced from popular JavaScript projects *Express*, *Commander.js*,*Javascript Algorithms*, and *Lodash*.

Our results show that when comparing DynaSPEA-II to baseline SPEA-II, DynaSPEA-II achieves higher coverage in 13.9% of classes, with an average increase of 4.92% for classes where a statistically significant difference was found. DynaSPEA-II and DynaMOSA have equal performance, with no statistically significant difference being found between the two algorithms.

This paper adds to the literature with the following contributions:

(1) The conceptual adaptation of the DynaMOSA features to the SPEA-II algorithm.
(2) The implementation of SPEA-II and DynaSPEA-II in the SynTest-Core framework.
(3) An empirical study, and analysis of the chosen algorithms.
(4) A full replication package for the empirical evaluation [2].

The remainder of the paper will be structured in the following way. Chapter 2 presents the background, detailing important concepts needed to understand the paper, such as evolutionary intelligence and search-based software testing. Chapter 3 details the approach, discussing how the SPEA-II algorithm was adapted to test case generation, and modified into DynaSPEA-II. The study design setup and analysis of the results are found in chapters 4 and 5 respectively. Chapter 6 addresses the threats to validity, followed by Chapter 7 which discusses the responsible research. Chapter 8 concludes the paper and addresses future research that can be performed.

2 BACKGROUND

This chapter provides the background information necessary to understand the research context and concepts. In subsection 2.1, we discuss the concept of evolutionary intelligence, explain the mechanisms of SPEA-II, and its differences with NSGA-II. The subsequent subsection focuses on the domain of test case generation, explaining techniques, such as Search-based Software Testing, and a notable algorithm within this category, DynaMOSA.

2.1 Evolutionary Intelligence

Evolutionary algorithms (EAs) are algorithms based on the concepts of natural evolution and 'survival of the fittest' [22]. EAs

[1] https://github.com/syntest-framework/syntest-javascript

[2] https://github.com/Chao-Ran-Erwin/CSE3000-Replication-Package

simulate evolution by iteratively evolving a group of candidate solutions, known as a *population*. Every individual solution is given a *fitness value* that quantifies how high the quality of the individual is. The solutions with better fitness values are given a higher likelihood of being picked as parents, to simulate the survival of the fittest principle. Evolution is mimicked by making individuals undergo various variation operations, such as crossover, and mutation to create new solutions.

Multi-objective evolutionary algorithms (MOEAs) address the challenge of optimizing multiple conflicting objectives simultaneously. Since there are multiple objectives, individuals cannot be compared based on a single fitness number. Instead, they are assessed based on *Pareto dominance*. An individual dominates another if it is better in one or multiple objectives, and equal in all others. The most valuable solutions are the *Pareto optimal* solutions, also known as non-dominated solutions. These solutions are not dominated by any other, with the set of all non-dominated solutions forming the *Pareto front*. The goal of multi-objective evolutionary algorithms is essentially to approximate this Pareto front as well as possible.

SPEA-II is an elitist multi-objective optimization algorithm, proposed by Zitzler et al [23]. SPEA-II aims to approximate the Pareto front by iteratively evolving a population and storing the best individuals in an external archive. It starts by initializing a random population and an empty external archive (lines 2 and 3 of algorithm 1). While the stopping criterion has not been met, the population evolves to find better solutions. The fitness of each solution in the population and archive is calculated (line 5). The fitness formula of SPEA-II consists of two parts raw fitness ($R(i)$) and density information ($D(i)$). $R(i)$ is calculated by summing the *strength* of a solution's dominators, with a solution's strength being the number of individuals it dominates. Therefore, fitness is to be minimized, with a non-dominated solution having a raw fitness of 0 and a dominated solution having a high fitness. Density information is added to the formula in order to increase the diversity of solutions. Density information is necessary, as when there are many non-dominated solutions, their raw fitness score will be identical, making the search equivalent to a random search. The density estimation technique used in SPEA-II is an adaptation of the k−th nearest neighbor method. For each individual, the distance to the k−th nearest neighbor is calculated, denoted as σ_i^k. Since fitness is to be minimized, the inverse of the distance is taken, leading to the formula for $D(i)$ being:

$$D(i) = \frac{1}{\sigma_i^k + 2} \tag{1}$$

Two is added to the denominator to avoid dividing by zero and to ensure that $D(i) < 1$, so that non-dominated solutions have a fitness of < 1. This leads to the final fitness formula of an individual i being:

$$F(i) = R(i) + D(i) \tag{2}$$

After the fitness calculation is finished, the algorithm proceeds with selecting the mating pool, also known as *environmental selection* (lines 6 to 12 of algorithm 1). First, all non-dominated solutions are added to the archive. If the archive is not full, the remaining solutions are sorted in ascending order of fitness and added to the archive until it is full. If there are too many solutions, the truncation method is called. Solutions are removed which are most similar to one another. Similarity is decided via the same k−th nearest neighbor technique used in the calculation of $D(i)$. Hence, the individual with the smallest σ_i^k is removed until the number of solutions equals the archive size. After the archive's population has been established, it will serve as the mating pool from which new offspring will be generated (line 13).

Algorithm 1: SPEA-II

Input :
$U = \{u_1, \ldots, u_m\}$ the set of coverage targets of a program
Population size N
Archive size $\overline{N}$
Output : A test suite T

1 **begin**
2 $P_0 \leftarrow$ RANDOM-POPULATION(N)
3 $\overline{P_0} \leftarrow \emptyset$
4 **while** *not (search_budget_consumed)* **do**
5 $F_t \leftarrow$ CALCULATE-FITNESS($P_t \cup \overline{P_t}$)
6 $\overline{P_t} \leftarrow$ ADD-NONDOMINATED(F_t)
7 **if** $|\overline{P_t}| < \overline{N}$ **then**
8 $\overline{P_t} \leftarrow \overline{P_t} \cup$ BEST-DOMINATED(F_t)
9 **end**
10 **if** $|\overline{P_t}| > \overline{N}$ **then**
11 $\overline{P_t} \leftarrow$ TRUNCATION($\overline{P_t}$)
12 **end**
13 $P_t \leftarrow$ GENERATE-OFFSPRING($\overline{P_t}$)
14 $t \leftarrow t + 1$
15 **end**
16 $T \leftarrow \overline{P_t}$
17 **return** T;

Non-dominated Sorting Genetic Algorithm II (NSGA-II) is a non-dominated sorting based evolutionary algorithm for multi-objective optimization [6]. It is very similar to SPEA-II with a few key differences. Instead of using the concept of strength, it chooses the best test cases via the *FAST-NONDOMINATED-SORT* algorithm. This algorithm finds the current test cases forming the Pareto front, and adds them to the next generation. This is repeated until the population size is reached. Furthermore, NSGA-II uses *crowding-distance* in order to maintain a more diverse solution set, while SPEA-II uses the k-th nearest neighbor technique. Once the population size has been reached, it is evolved via the same operators as SPEA-II.

2.2 Test Case Generation

Tests have always been necessary, yet laborious to write, therefore many techniques have been proposed over the years to automate this process. Examples being, symbolic execution [5], feedback-directed random generation [12], and search-based software testing (SBST) [10], with the latter being the category SPEA-II falls in.

Search-based Software Testing (SBST) is the practice of using a meta-heuristic optimization search technique, for instance Genetic

Algorithms, to generate test cases [11]. The search is guided by a fitness function, which can calculate the distance to the objectives, and thereby assess solutions. The goal of SBST is to evolve test cases in order to reduce the distance to and eventually cover all objectives.

In the context of test generation as a many-objective optimization problem, objectives are the individual distances to all test targets under test [13]. The definition of distance depends on the selected coverage type. For example, the distance for branch coverage consists of *approach level* [20] and *branch distance* [7]. Approach level is a heuristic that measures the number of control dependencies between the execution trace and target branch, with a lower approach level indicating a more direct path to the objective. Branch distance finds the conditional expression where the execution diverges from the target and calculates the distance to fulfilling the expression. Examples of meta-heuristic search algorithms are *Many Independent Objective* (MIO) [3], *Whole Suite with Archive* (WSA) [17], and DynaMOSA [14]), with studies showing that DynaMOSA is more effective than its peers for Java [15] and Python [9] classes.

Dynamic Many-Objective Sorting Algorithm (DynaMOSA) is an extension of NSGA-II that incorporates domain knowledge to make the algorithm more suitable towards automatic test case generation. DynaMOSA is an improved version of *Many-Objective Sorting Algorithm* (MOSA), which added *preference sorting*, and *archiving* to NSGA-II. DynaMOSA improved on MOSA by adding *dynamic target selection*.

MOSA uses preference sorting in order to favor extreme solutions that are closest to one or more objectives, over trade-off solutions that are non-dominated due to being good in multiple objectives. For instance, the vector $[0, 2]$, would be favored over $[1, 1]$ as it is closest to covering an objective. These solutions get prioritized, because in the context of test generation, the only solutions that are relevant, are ones that actually cover one or more targets. The preference criterion is necessary, as NSGA-II has scalability issues with solving problems with more than three objectives [8]. This is due to the number of non-dominated solutions growing exponentially with the number of targets. Hence, sorting based on the property of Pareto dominance becomes ineffective, making the search process akin to a random search. SPEA-II shares this same flaw, and will therefore also need preference sorting in order to efficiently generate test cases. MOSA maintains an archive which stores the best solution for each objective, which also serves as the final test suite that is returned. By maintaining this archive, the search can be focused on the remaining uncovered targets, this however is a flaw of MOSA.

The problem with optimizing for all uncovered objectives, is that it includes unreachable ones. DynaMOSA addresses this by adding dynamic target selection, meaning that the objectives to be covered are constantly changing. Branches are often dependent on each other, for example nested if statements, the dependent branch can only be covered once the first branch is covered. Therefore, with dynamic target selection, the dependent branch will only be optimized for once it can be reached.

3 APPROACH

This chapter will discuss how the SPEA-II algorithm was adapted from a numerical context into the test case generation domain, and how it was modified into the DynaSPEA-II algorithm.

3.1 SPEA-II Adaptation to Test Case generation

The baseline version of SPEA-II had to be changed in order to make it suitable for test case generation. In this paper, we approach automatic test case generation as a many-objective optimization problem, following the concept proposed by Panichella et al. [4]. We consider the distance to each test target as an individual objective to be optimized. Thus, a candidate solution is a singular test case, whose fitness is defined as a vector of k values, with each value representing the distance to an objective. We use these distances to perform the fitness calculation of SPEA-II, that was defined in chapter 2. For the baseline version of SPEA-II, solutions are being compared on every objective to determine their raw fitness and diversity.

3.2 DynaMOSA Features Adaptation

The three DynaMOSA features, test archiving, preference criterion, and dynamic target selection were added to SPEA-II to create DynaSPEA-II, which can be seen in algorithm 2. The changes from the baseline version have been highlighted in orange, red, and blue respectively.

Archiving has been added, via an additional external archive. This archive stores the solutions which are closest to one or multiple objectives and returns it as the final test suite. Meaning, that the new algorithm contains two external archives. In order to avoid confusion, the original SPEA-II archive will be referred to as the *SPEA-archive*, while the new archive will be called the *MOSA-archive*. The two archives serve distinct purposes. The former stores the best solutions across generations, to serve as the mating pool for the next generation, while the latter is not directly involved in the evolutionary process but is maintained to guide the search towards uncovered targets. The MOSA-archive functions via the $UPDATE-ARCHIVE$ method. $UPDATE-ARCHIVE$ takes a population and updates the archive to store the best cases for every objective, picking the shortest test solution in case of a draw. It starts off by evaluating the initial population (line 6 of algorithm 2) and is called again every time new solutions are created (line 20).

Preference criterion is incorporated by modifying the environmental selection method. The new environmental selection starts by finding all solutions closest to one or more objectives and storing them in $frontZero$ (line 8 of algorithm 2). Moreover, all test cases in $frontZero$ are assigned rank 0 to increase their likelihood of being picked as parents. If the length of $frontZero$ is less than the size of the SPEA-archive, normal SPEA-II environmental selection is performed with the $remainingPopulation$ and $remainingSize$ (lines 11 to 18 in algorithm 2). The $remainigPopulation$ variable is declared as the union of the population and the SPEA-archive, minus all solutions in $frontZero$ (line 9), and $remainingSize$ as the difference between the archive size and length of $frontZero$ (line 10). The final mating pool is the union between $frontZero$ and the results of the baseline SPEA-II environmental selection, from which offspring are generated (line 19)

Dynamic target selection is the final addition to our algorithm, meaning that the search process now only optimizes for objectives that are reachable. This is done via the *control dependency graph (CDG)*, which determines which targets are independent of others. The subset of independent targets get initialized as $U*$ (line 6 of algorithm 2) and is updated once new test cases are generated via the $UPDATE-TARGETS$ method (line 21). The methods affected by this feature are $CALCULATE-FRONT-ZERO$, and $CALCULATE-FITNESS$. It is important to note that dynamic target selection, does not directly generate better tests, as the same individuals will be selected with or without this feature in the affected methods. The search will, however, complete faster as there are fewer objectives to evaluate. Given the limited time resources, this can potentially indirectly lead to higher coverage, as more time can be spent on evolving the population.

Algorithm 2: DynaSPEA-II

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

1 **begin**
2 $U* \leftarrow$ targets in U with no control dependencies
3 $P_0 \leftarrow$ RANDOM-POPULATION(N)
4 $\overline{P_0} \leftarrow \emptyset$
5 MOSA-archive $\leftarrow$ UPDATE-ARCHIVE($P_t, \emptyset$)
6 $U* \leftarrow$ UPDATE-TARGETS($U*, G, \phi$)
7 **while** *not (search_budget_consumed)* **do**
8 $Z \leftarrow$ CALCULATE-FRONT-ZERO($P_t \cup \overline{P_t}, U*$)
9 $R \leftarrow ((P_t \cup \overline{P_t}) \setminus Z)$
10 $S \leftarrow \overline{N} - |Z|$
11 $F_t \leftarrow$ CALCULATE-FITNESS($R, U*$)
12 $\overline{P_t} \leftarrow$ ADD-NONDOMINATED(F_t)
13 **if** $|\overline{P_t}| < S$ **then**
14 $\overline{P_t} \leftarrow \overline{P_t} \cup$ BEST-DOMINATED(F_t)
15 **end**
16 **if** $|\overline{P_t}| > S$ **then**
17 $\overline{P_t} \leftarrow$ TRUNCATION($\overline{P_t}$)
18 **end**
19 $P_t \leftarrow$ GENERATE-OFFSPRING($\overline{P_t} \cup Z$)
20 $MOSA-archive \leftarrow$ UPDATE-ARCHIVE(P_t, MOSA-archive)
21 $U* \leftarrow$ UPDATE-TARGETS($U*, G, \phi$)
22 $t \leftarrow t + 1$
23 **end**
24 $T \leftarrow$ MOSA-archive
25 **return** T;

4 STUDY DESIGN

This chapter details how we designed and carried out the experiment to determine how well SPEA-II and DynaSPEA-II perform in automatic test case generation.

4.1 Research Questions

In order to gauge the performance of DynaSPEA-II, and the DynaMOSA enhancements, the experiment will answer the following questions:

- *RQ_1: How does DynaSPEA-II perform compared to DynaMOSA on branch coverage?*
- *RQ_2: How effective are the additions of DynaMOSA features to SPEA-II?*

4.2 Configurations

In order to answer the research questions, the following pairs of algorithms will be compared.

(1) DynaSPEA-II vs. DynaMOSA.
(2) DynaSPEA-II vs. SPEA-II.
(3) MOSASPEA-II vs. SPEA-II
(4) DynaSPEA-II vs. MOSASPEA-II

The first comparison will answer the first research question, by comparing the novel DynaSPEA-II against the state-of-the-art DynaMOSA. The second research question will be answered by pairs two through four. By comparing all SPEA-II variants, we can see how impactful each individual enhancement is, and the total effect. SPEA-II is the baseline version of the algorithm that was adapted to case generation (depicted in algorithm 1), and searches while considering every objective. DynaSPEA-II is the algorithm shown in algorithm 2's pseudocode, it is SPEA-II with all DynaMOSA enhancements, including preference sorting, an additional archive, and dynamic target selection. MOSASPEA-II is the same algorithm as DynaSPEA-II, but rather than only considering the objectives that are uncovered and independent of others, it considers *all* uncovered targets. This variant will also be tested to see the impact of dynamic target selection.

4.3 Prototype

To answer the research questions, SPEA-II, MOSASPEA-II and DynaSPEA-II have been implemented in the SynTest-Core[3] framework, which also already contains the implementation for DynaMOSA. The algorithms will be evaluated via the SynTest-JavaScript-Benchmark[4] repository.

[3] https://github.com/syntest-framework/syntest-core
[4] https://github.com/syntest-framework/syntest-javascript-benchmark

4.4 Benchmark

The dataset of SynTest-JavaScript-Benchmark consists of classes from five different JavaScript projects, namely *Express*[5], *Commander.js*[6], *Moment.js*[7], *JavaScript Algorithms*[8], and *Lodash*[9]. This benchmark has been used in literature before for unit-level test case generation [18], and has been visually inspected to ensure it is diverse and of high quality. The projects were chosen based on their popularity, measured in GitHub stars. From these projects, classes with a Cyclomatic Complexity ≥ 2 were selected. For our experiment, we removed some files that were causing problems when running the experiment. Specifically, the entire Moment project, *application.js* from Express and *dijkstra.js* from JavaScript-Algorithms were omitted, leaving 36 remaining classes.

4.5 Parameter Settings

For this experiment, we have chosen to employ the default parameters used in similar experiments [14] [18], resulting in the following selection:

- Population size: 50 individuals
- SPEA-Archive size: 50 individuals
- Crossover: single-point crossover with crossover probability of 0.75
- Mutation: uniform mutation with mutation probability of $1/n$, with n being equal to the number of statements in the test case.
- Selection: tournament selection with size set to 10
- Search timeout: 60 seconds

The population- and archive size are less than the default value of `SPEA-II` in numeric experiments [23], because search time is of paramount importance in our experiment. Since we are comparing test cases rather than numbers, it takes longer to evaluate a population. The larger the population, the more time it takes to evaluate one generation, leading to less evolution. Therefore, we have chosen to opt for a smaller population- and archive size. These hyperparameters were used for all algorithms, with DynaMOSA lacking the SPEA-archive parameter as it is not part of that algorithm.

4.6 Experimental Protocol

To answer our research questions, this experiment ran every algorithm on every class and compared the results. There were six algorithms (`SPEA-II`, MOSASPEA-II, DynaSPEA-II, `NSGA-II`, MOSA, DynaMOSA) and 36 classes for a total of 216 instances. One run has a search budget of 60 seconds, with another ~30 seconds for pre- and post-processing. This results in one run of the experiment taking (216 instances $\times$ 90 s)/(60s $\times$ 60s) $\approx$ 5.4 hours of consecutive computation time. Additionally, to counter the inherent randomness in automatic test case generation, the experiment was run ten times. Leading to a final sum of (5.4 h $\times$ 10 runs) $\approx$ 54 hours. The algorithms that were run but not mentioned in section 4.2 *Configurations*, are used as baselines to assess whether the algorithms have been correctly implemented (*e.g.*,`NSGA-II` should perform worse than DynaMOSA). The experiment was performed on a system with 2 AMD EPYC 7H12 (64 core, 3293.082 MHz, 256 threads) processors with 512 GB of RAM, running 100 cores in parallel.

The results will be statistically analyzed to see if one algorithm is statistically superior to another. This will be done by first applying the non-parametric Wilcoxon test with a p-value of 0.05 [21], to determine if there is a statistically significant difference. To measure the effect size, the Vargha-Daleney $\hat{A}_{12}$ statistic [19] is used.

[5]https://expressjs.com/
[6]https://tj.github.io/commander.js/
[7]https://momentjs.com/
[8]https://github.com/trekhleb/javascript-algorithms
[9]https://lodash.com/

5 RESULTS

This chapter discusses and analyses the results, in order to answer the research questions stated in Chapter 4. In the following sections, when comparing the code coverage of two algorithms, we refer to the difference in percentage points.

5.1 RQ1: How does DynaSPEA-II perform compared to DynaMOSA on branch coverage?

In Table 1, we report the outcomes of our experiment by showing the median branch coverage and Inter-Quartile-Range (IQR) for every class and every algorithm. The largest values per row have been highlighted in gray, with no cells being highlighted if all values are equal. Note that nine files are missing from the table, all from JavaScript-Algorithms, namely *articulationPoints.js*, *bellmanFord.js*, *bfTravellingSalesman.js*, *detectDirectedCycle.js*, *detectUndirectedCycle.js*, *eulerianPath.js*, *floydWarshall.js*, *hamiltonianCycle.js*, and *stronglyConnectedComponents.js*. No algorithm managed to cover any branches, because they required graphs as an input parameter, which are very difficult to generate. The classes have, therefore, been omitted from the table, but are still taken into account when calculating statistics across all classes.

Table 1 shows that DynaSPEA-II has a median branch coverage of 40.5%, and DynaMOSA achieves a coverage of 39.9%. The file with the largest difference, in favor of DynaSPEA-II, is *depthFirstSearch.js* with a 16.7% increase on the median. On the other hand, *breadthFirstSearch.js* shows the largest gap on the side of DynaMOSA, with a 6.3% difference.

An inspection of Table 1 gives the idea that DynaSPEA-II and DynaMOSA are generally equals. The median branch coverage across the entire benchmark only differs $\approx$ 0.006 and for most classes the scores are almost identical. This intuition is proven by Table 2, which displays the result of the non parametric Wilcoxon test and Vargha-Daleney $\hat{A}_{12}$ statistic. The *#Win* column indicates the number of times when the left algorithm shows a statistically significant improvement compared to the right algorithm. The *No diff.* column represents the number of instances where there is no statistically significant difference to suggest a distinction between the two algorithms. Lastly, the "#Lose" column shows the number of instances where the left algorithm has statistically worse results than the right algorithm. Additionally, the "#Win" and "#Lose" columns also include the $\hat{A}_{12}$ effect size, which is categorized as *Small*, *Medium*, *Large*, or *Negligible*. With the boundaries for these categories being

- A12 $\leq$ 0.56: Negligible effect size
- 0.56 < A12 $\leq$ 0.64: Small effect size
- 0.64 < A12 $\leq$ 0.71: Medium effect size

- A12 > 0.71: Large effect size

Table 2 shows that for all 36 classes, there is no statistically significant difference between DynaSPEA-II and DynaMOSA in the number of branches covered, meaning that the algorithms have equal performance.

5.2 RQ2 How effective are the additions of DynaMOSA features to SPEA-II?

The results for all variants of SPEA-II are shown in Table 1, with SPEA-II having a median branch coverage of 39.4%, and MOSASPEA-II and DynaSPEA-II having an equal coverage of 40.5%. Comparing SPEA-II to DynaSPEA-II, the latter has a higher median coverage in five files, ranging from 2.2% to 16.7% with an average increase of 4.92%. The smallest increase manifests in *utils.js*, with the largest occurring in *depthFirstSearch.js*. The only decrease in performance takes place in *breadthFirstSearch.js*, going from 18.8% to 12.5%. From Table 2 we observe that DynaSPEA-II is significantly better than SPEA-II in five classes, those being the ones where DynaSPEA-II has higher median branch coverage. This results in a significant increase in branch coverage in 13.9% of classes. For the other 31 classes, there is no statistically significant difference. These improvements can be attributed to the DynaMOSA features that were added to SPEA-II to optimize it for the domain of test case generation, as the other variables remained the same.

To isolate the effects of the preference criterion in combination with the MOSA-archive, and dynamic target selection, Table 2 also compares MOSASPEA-II with SPEA-II and DynaSPEA-II. From Table 2 we can discern that there is no significant difference between MOSASPEA-II and DynaSPEA-II for any class, indicating that dynamic target selection does not aid in improving branch coverage. However, when comparing the results from MOSASPEA-II vs. SPEA-II to DynaSPEA-II vs. SPEA-II we see that there is a difference of two classes, those being *help.js* and *depthFirstSearch.js*. Hence, dynamic target selection has assisted in a significant manner.

6 THREATS TO VALIDITY

This chapter discusses potential concerns to the validity of the research, and what measures were taken in order to address these threats.

Construct validity refers to how well the measure of a construct, computes the intended concept. In our case, how well branch coverage (construct), computes algorithm performance (concept). Branch coverage is a widely used metric in the literature [14], and gives a reasonable estimation of an algorithm's effectiveness in test case generation.

Internal validity refers to the extent that our findings are a result of manipulated variables, rather than confounding factors. The first potential threat is randomness. The stochastic nature of evolutionary algorithms can lead to findings that do not accurately reflect reality. In order to combat this, we repeated the experiment 10 times and took the average to get more precise results. Another potential threat is in our choice of hyperparameters. We used default values used in literature in similar experiments [18]. Furthermore, previous research has shown that while adjusting these parameters can affect the performance of the search algorithm, the default values yield reasonable outcomes [4].

External validity refers to whether the findings of our study are generalizable and can be applied to other contexts. Our benchmark consists of 36 classes from four different projects. These classes are diverse in multiple factors, such as size, purpose, and syntax. However, generalizability can be improved by increasing the size of the benchmark.

Conclusion validity refers to the degree that the conclusions we reach are reasonable. We ran all algorithms on the same system, to create a controlled environment. Furthermore, every algorithm was run 10 times on different seeds to ensure that there was no one seed where a particular algorithm had an advantage. In terms of analysis, we used the non-parametric Wilcoxon test and Vargha-Daleney $\hat{A}_{12}$ statistic to measure statistical significance and effect size respectively. All our conclusions are based on these tests showing statistically significant results.

7 RESPONSIBLE RESEARCH

This chapter outlines the ethical considerations of the project and the reproducibility of the experiment.

7.1 Research Integrity

The goal of this research is to investigate how well SPEA-II performs with the enhancements of DynaMOSA, **not** to replace DynaMOSA. If it had better performance, this would have been a nice bonus, but it was not the primary objective. The main goal was to fill the gap in the literature. Therefore, there is no incentive to falsify the results, or bias the dataset to get more favorable results. Additionally, the dataset was sampled from multiple different sources in order to make it as diverse and unbiased as possible.

7.2 Reproducibility

Automatic test case generation is inherently a stochastic process, so it is improbable that the exact same results will be acquired when reproducing the experiment. To combat this, however, multiple runs were performed, and the average result was taken. Therefore, the results should be comparable when reproducing the experiment on similar equipment. It is important to note that the specifications of the machine where the experiment is run on, can greatly vary the results. Hence, the optimal way to reproduce the empirical evaluation is to achieve an environment that is as similar as possible. A replication package has been provided to help others recreate the experiment.

8 CONCLUSION AND FUTURE WORK

In this paper, we explored the potential of SPEA-II for automatic test case generation. This was done by adapting the algorithm to make it suitable for test case generation, and thereafter enhancing it with features from the state-of-the-art algorithm DynaMOSA. We conducted an empirical experiment, testing multiple algorithms on a diverse benchmark consisting of 36 JavaScript classes, sampled from 4 popular JavaScript projects. Our results show that the DynaMOSA modifications to SPEA-II significantly improve the branch coverage in 13.9% of classes, with an average increase of

Benchmark	File Name	SPEAII		MOSASPEAII		DynaSPEAII		DynaMOSA	
		Median	IQR	Median	IQR	Median	IQR	Median	IQR
Commander.js	help.js	0.470	0.015	0.492	0.015	0.500	0.000	0.500	0.000
	option.js	0.500	0.000	0.500	0.000	0.500	0.000	0.500	0.000
	suggestSimilar.js	0.719	0.023	0.719	0.000	0.719	0.000	0.719	0.000
Express	query.js	0.667	0.000	0.667	0.000	0.667	0.000	0.667	0.000
	request.js	0.326	0.000	0.326	0.000	0.326	0.000	0.326	0.000
	response.js	0.163	0.014	0.196	0.015	0.190	0.011	0.196	0.014
	utils.js	0.413	0.022	0.435	0.016	0.435	0.022	0.424	0.022
	view.js	0.375	0.000	0.375	0.000	0.375	0.000	0.375	0.000
JavaScript-Algorithms	breadthFirstSearch.js	0.188	0.125	0.125	0.125	0.125	0.094	0.188	0.125
	depthFirstSearch.js	0.000	0.000	0.167	0.167	0.167	0.167	0.000	0.167
	kruskal.js	0.200	0.000	0.200	0.000	0.200	0.000	0.200	0.000
	prim.js	0.167	0.000	0.167	0.000	0.167	0.000	0.167	0.000
	Knapsack.js	0.575	0.000	0.575	0.000	0.575	0.000	0.575	0.000
	KnapsackItem.js	0.500	0.000	0.500	0.000	0.500	0.000	0.500	0.000
	Matrix.js	0.079	0.000	0.079	0.000	0.079	0.000	0.079	0.000
	CountingSort.js	0.571	0.000	0.571	0.000	0.571	0.054	0.571	0.000
	RedBlackTree.js	0.294	0.000	0.294	0.000	0.294	0.000	0.294	0.000
Lodash	equalArrays.js	0.833	0.000	0.833	0.000	0.833	0.000	0.833	0.000
	hasPath.js	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000
	random.js	1.000	0.054	1.000	0.000	1.000	0.000	1.000	0.000
	result.js	0.800	0.000	0.800	0.000	0.800	0.000	0.800	0.000
	slice.js	0.950	0.038	1.000	0.000	1.000	0.000	1.000	0.000
	split.js	0.875	0.000	0.875	0.000	0.875	0.000	0.875	0.000
	toNumber.js	0.650	0.000	0.650	0.000	0.650	0.000	0.650	0.000
	transform.js	0.875	0.083	0.833	0.167	0.875	0.146	0.917	0.146
	truncate.js	0.559	0.000	0.559	0.000	0.559	0.000	0.559	0.000
	unzip.js	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000
Median across all classes		0.394		0.405		0.405		0.399	

Table 1: Median branch coverage and Inter-Quartile-Range per algorithm, with the cells of the largest values highlighted in gray.

Comparison	#Win				#No diff	#Lose			
	Negl.	Small	Medium	Large	Negl.	Negl.	Small	Medium	Large
DynaSPEA-II vs. DynaMOSA	-	-	-	-	36	-	-	-	-
DynaSPEA-II vs. SPEA-II	-	-	-	5	31	-	-	-	-
MOSASPEA-II vs SPEA-II	-	-	-	3	33	-	-	-	-
DynaSPEA-II vs MOSASPEA-II	-	-	-	-	36	-	-	-	-

Table 2: Results of statistical analysis w.r.t. branch coverage

4.92%, and that the DynaSPEA-II algorithm showed equal performance to DynaMOSA.

While this study has provided valuable insights into the performance of SPEA-II in generating tests for JavaScript classes, there are several avenues to explore for future work to expand upon these findings. Firstly, broadening the scope of the benchmark by including more files from different sources. A more expansive benchmark leads to increased reliability, and generalizability of the results. Secondly, investigating the impact of different hyperparameters on the performance of the algorithm. Due to time constraints, we chose the default parameters, but conducting hyperparameter experiments can potentially yield improved results. Lastly, expanding the investigation to different programming languages, specifically static ones. In our research, we only looked at the performance of SPEA-II for JavaScript. Testing our novel algorithm in a static language, for instance Java, can produce valuable insights on how

generalizable the performance of SPEA-II is across different types of programming languages.

Overall, this research contributes to the field of automatic test case generation and opens up new possibilities for future research to explore and refine the SPEA-II algorithm for generating tests in various contexts.

REFERENCES

[1] 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)*. IEEE, 263–272.

[2] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying search based software engineering with Sapienz at Facebook. In *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*. Springer, 3–45.

[3] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206.

[4] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18 (2013), 594–623.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[6] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[7] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.

[8] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. 2015. Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)* 48, 1 (2015), 1–35.

[9] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for python. *Empirical Software Engineering* 28, 2 (2023), 36.

[10] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[11] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.

[12] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[13] 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)*. IEEE, 1–10.

[14] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[15] 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.

[16] 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*. 547–558.

[17] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22 (2017), 852–893.

[18] 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: 14th International Symposium, SSBSE 2022, Singapore, November 17–18, 2022, Proceedings*. Springer, 67–82.

[19] András Vargha and Harold D Delaney. 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), 101–132.

[20] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and software technology* 43, 14 (2001), 841–854.

[21] CONOVER WJ. 1998. "Practical nonparametric statistics. vol. 350.

[22] Xinjie Yu and Mitsuo Gen. 2010. *Introduction to evolutionary algorithms*. Springer Science & Business Media.

[23] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK-report* 103 (2001).