Performance of the Pareto Envelope-Based Search Algorithm - II in Automated Test Case Generation

Apoorva Abhishek[1]

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

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

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

ABSTRACT

Software testing is an important yet time consuming task in the software development life cycle. Artificial Intelligence (AI) algorithms have been used to automate this task and have proven to be proficient at it. This research focuses on the automated testing of JavaScript programs, and builds upon the existing SynTest framework that is the current state of the art, with the Dynamic Many Objective Sorting Algorithm (DynaMOSA) being the best performing AI algorithm for test case generation. DynaMOSA uses the Non-Dominated Sorting Algorithm - II (NSGA-II) as its base algorithm, and adds modifications to it. This paper investigates whether the use of the Pareto Envelope Based Search Algorithm - II (PESA-II) as the base algorithm results in improved performance. The contributions of this research includes a modified PESA-II integrated into the SynTest framework, using inspiration from DynaMOSA. Moreover, we answer the question "How does the modified PESA-II perform compared to DynaMOSA in generating test cases for JavaScript programs?" The performance of the algorithms is measured based on the (branch and method) coverage of the test cases generated for a suite of JavaScript classes. The results show that the modified version of PESA-II outperforms the base version. However, neither manage to outperform DynaMOSA.

1 INTRODUCTION

Software testing is an essential part of the software development process. Developers spend a significant amount of time in writing test cases for their code. To streamline this process, numerous strategies such as symbolic execution, model-based test case generation and search-based testing have been devised to automate test case generation [1]. Search-based testing with Artificial Intelligence (AI) algorithms not only automate test case generation but also enhance code coverage and bug detection capabilities [9].

This thesis focuses on the automated testing of JavaScript programs using search-based testing techniques. SynTest-JavaScript [11] is a software tool that generates unit tests for JavaScript programs. It uses many-objective evolutionary algorithms (MaOEA) to navigate the search space in order to generate the test cases. The Dynamic Many Objective Sorting Algorithm (DynaMOSA) [9] is one such evolutionary algorithm that has proven to be very effective in generating optimal test cases. We will explore other adaptations of evolutionary algorithms using inspiration from DynaMOSA which could potentially yield better results and improve the current state of this field.

None of the generic many-objective algorithms can scale to the number of objectives typically found in coverage testing [9]. DynaMOSA uses the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [5] as its base algorithm and modifies it to make it better suited to the test case generation problem. We will find out whether the choice of NSGA-II as the base algorithm was the right one, or if there is a better alternative.

The aim of this research is to explore the Pareto Envelope-based Evolutionary Algorithm - II (PESA - II) [4] as the base algorithm in generating test cases for JavaScript programs. The key feature of PESA-II is region based selection which aims to create diversity on the estimated Pareto front. This is done to generate a diverse population in the hopes of creating a more accurate estimation of the actual Pareto Front. However, PESA-II is known to perform well with only a small number of objectives. In the test case generation problem, we are dealing with 100s of objectives and PESA-II does not scale well with so many objectives. This is why we have augmented the features from DynaMOSA to improve performance in generating test cases.

The main question we seek to answer is: *How does PESA-II augmented with DynaMOSA features perform in generating test cases for JavaScript programs compared to DynaMOSA*? To make PESA-II better suited for the problem of test case generation, we have augmented features from DynaMOSA such as:

- Selection with the use of the *Preference Criterion* mentioned in [9].
- Dynamic Selection of the optimization targets, also mentioned in [9].

To compare the performance of the algorithms, we will use the SynTest-Benchmark [11] tool. It consists of a diverse set of JavaScript classes that the algorithms will generate test cases on. We will measure the quality of the test cases generated based on branch and method coverage.

The contributions of this research include:

- An implementation of the PESA-II algorithm adapted to the problem of test case generation. .
- An implementation of PESA-II augmented with DynaMOSA features - DynaPESA-II.
- A replication package that contain the results and statistical analysis scripts.

The implementations are integrated into the SynTest framework so that they can be used to generate test cases for JavaScript programs.

The remainder of this paper is structured as follows: Section 2 provides background knowledge about the problem of test case generation, the use of Evolutionary Algorithms for this problem, and an overview of the theory behind DynaMOSA and PESA-II. Section 3 details the base PESA-II implementation and the modifications made to the algorithm, enabling it to generate optimal test cases. Sections 4 and 5 present the method of evaluation and the results obtained. Sections 6 and 7 discuss any potential gaps in our research that could invalidate it and the ethical implications of this research, respectively. Finally, Section 8 concludes the research and provides insight into the possible future work.

2 BACKGROUND

The section will begin by providing an explanation of the concepts behind Evolutionary Algorithms and their application in the domain of search-based test case generation. Subsequently, we will delve into the theory of DynaMOSA, emphasizing its suitability for addressing the challenges associated with test case generation. Furthermore, we will explain the working of PESA-II.

2.1 Many-Objective Evolutionary Algorithms

A many-objective problem is formulated as:

$$\text{minimize} \quad F(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})) \tag{1}$$

$$\text{subject to} \quad \mathbf{x} \in S, \tag{2}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ is a solution in the search space S and m is the number of objectives [7]. This problem can be optimized by approximating the Pareto Front which is defined as follows:

Definition 2.1 (Pareto Front). Let $F(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))$ be a vector of m objective functions to be minimized, where $\mathbf{x} \in S$ is a decision vector from the feasible region S. The Pareto front, denoted PF, is the set of all non-dominated solutions in the objective space, defined as

$$\begin{aligned} PF = \{\mathbf{y} \in \mathbb{R}^m \mid \mathbf{y} = F(\mathbf{x}) \\ &\text{and there does not exist } \mathbf{x}' \in S \\ &\text{such that } F(\mathbf{x}') \preceq F(\mathbf{x}) \\ &\text{and } F(\mathbf{x}') \neq F(\mathbf{x})\}. \end{aligned}$$

Here, $\mathbf{y} = F(\mathbf{x})$ represents a point in the objective space, and $F(\mathbf{x}') \preceq F(\mathbf{x})$ means that $F(\mathbf{x}')$ is objective-wise less than or equal to $F(\mathbf{x})$. [12]

Evolutionary Algorithms (EAs) have gained prominence as powerful tools for tackling Many-Objective optimization problems, wherein the task involves optimizing more than three objectives simultaneously. As discussed in [2], EAs are inspired by the principles of natural selection and genetics. They are characterized as population-based stochastic algorithms that efficiently navigate through high-dimensional search spaces. In EAs, a set of candidate solutions, referred to as individuals, constitute a population. Each individual represents a feasible solution to the optimization problem. The aptitude or suitability of these solutions is measured using a fitness function, which quantifies the quality of each solution.

The typical workflow of EAs, depicted in Figure 1, involves several iterative steps. Initially, the algorithm starts with a randomly generated population of a fixed number of solutions. The individuals in this population undergo a *selection* process, where the probability of being selected is positively correlated with their fitness. The selected individuals are then subjected to genetic operators such as crossover (recombination) and mutation to generate offspring. These operators mimic natural genetic processes and introduce diversity and novelty into the population. The newly created offspring and the previous population are evaluated, and individuals are chosen for the next generation based on their fitness levels. This iterative process, often referred to as generations or iterations, continues until a predetermined termination criterion, such as a maximum number of generations or a satisfactory fitness level, is achieved.

2.2 MaOEAs in Search Based Testing

Search Based Software Testing (SBST) employs search algorithms and optimization techniques to automate the generation of test cases. It formulates testing as an optimization problem [6]. The adaptation of many-objective problems to test case generation has been covered in previous research. An overview of the modifications made to generic evolutionary algorithms are given below.

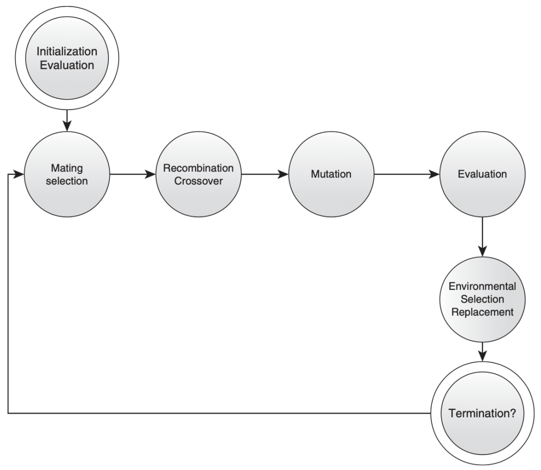

Figure 1: The Evolutionary Cycle [2]

The solutions in the population are represented as *encodings* which are the inputs to the methods under test. The objectives to be optimized are the individual distances from all the test targets in the class. The test targets are the branches and the objective values for the test targets are represented as the sum of the approach level (the number of control dependencies between the closest executed branch and the target branch) and the normalized branch distance. Each $f(x)$ in equation 1 is reformulated as: $f(t, x) = A(t, x) + d(t, x)$ where A represents the approach level, d represents the branch distance and t is the target branch [9].

The number of objectives usually exceeds that of typical problems that MaOEAs are used for. MaOEAs do not scale well with so many objectives since the number of solutions on the Pareto Front increase rapidly with an increase in the number of objectives. This is why DynaMOSA introduces certain features that tackle this issue.

2.3 DynaMOSA

To explain DynaMOSA, we will first explain the base algorithm NSGA-II, followed by the features implemented by MOSA and DynaMOSA.

2.3.1 NSGA-II [5]. NSGA-II employs a non-dominated sorting approach to classify the set of solutions into different fronts. The solutions that do not dominate each other constitute the first front (or Pareto Front), the next set of solutions which are only dominated by the individuals of the first front form the second front, and so on. During the environmental selection process, individuals are added to the next generations based on their rank (individuals from the first front are added, followed by the second, ...) until the limit for the next generation is reached. This mechanism allows the algorithm to identify and preserve Pareto-optimal solutions effectively.

NSGA-II introduces another novel component of the algorithm: the crowding distance assignment. For each solution, the crowding distance is a measure of how close its neighbors are in the objective space. A larger crowding distance means that the solution is surrounded by fewer neighbors. This is important because maintaining diversity prevents the algorithm from converging to a single solution, and instead helps in exploring various regions of the Pareto front. During the mating selection process, NSGA-II prefers individuals that have a lower rank and are less crowded.

2.3.2 Many Objective Sorting Algorithm (MOSA) [8]. The Many Objective Sorting Algorithm (MOSA) was designed with the specific objective of enhancing automated test case generation. It targets all uncovered branches at once by considering them as different objectives to be optimized in parallel. While NSGA-II is a more general-purpose multi-objective optimization algorithm, MOSA introduces certain mechanisms that make it particularly suited for test case generation.

As discussed earlier, MaOEAs (including NSGA-II) do not scale well with the number of objectives being used in test case generation. That is why MOSA replaces the traditional non-dominated sorting method with a new ranking algorithm based on the Preference Criterion defined in [8]. This method finds the solutions with the lowest fitness score for each of the objectives, and assigns them with the rank 0 (giving them a higher chance of survival for the next generation). The rest of the solutions are then ranked according to the regular non-dominated sorting method. It also uses an archive to store all test cases satisfying the previously uncovered branches. Once a branch is covered, it is ignored and removed from the set of objectives being optimized.

2.3.3 Dynamic Selection of Optimisation Targets. Suppose there is a target *t1* that has not been covered by any solutions in the current population. Let us also imagine that there are two additional targets, *t2* and *t3*, which have a structural dependency on *t1*. This means they cannot be properly addressed until *t1* is dealt with first. Refer to figure 2 for a clear understanding of the Control Flow Graph example.

The Preference Criterion comes into play here. It would prioritize two solutions with the lowest fitness values for *t2* and *t3*. However, these solutions offer no real benefit in terms of covering their respective test targets, *t2* and *t3*, since these targets cannot be reached until *t1* is covered [9].

To efficiently manage this situation, DynaMOSA uses a dynamic selection process for test targets. It excludes targets that are currently unreachable due to structural dependencies from the evaluation of the current population. In other words, it prioritizes dealing with the root issue, *t1* in this case, before moving onto issues that are dependent on it. This dynamic selection approach ensures that resources are not wasted on "redundant solutions" or test cases that cannot be resolved until another issue is addressed first. This allows DynaMOSA to converge to optimal solutions faster than the alternatives.

2.4 PESA-II [4]

PESA-II is a multi-objective evolutionary algorithm that has shown to perform well at optimizing problems with 2-3 objectives. It differs from other evolutionary algorithms mainly in its selection process. A crucial component of PESA-II is the maintenance of an archive of non-dominated solutions, which serves as a current estimation of the Pareto Front. This archive plays a vital role in the algorithm's operation, functioning as the source of solutions for selection and recombination.

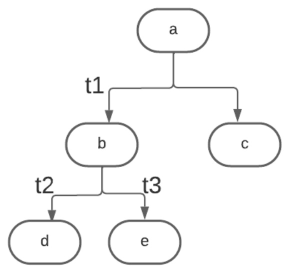

Figure 2: Example Control Flow Graph

Within this archive, PESA-II does not directly assign fitness values to individuals. Instead, it segments the objective space into discrete regions, referred to as 'hyper-boxes'. The fitness of each hyper-box is derived from the density of individuals it contains — the number of solutions from the archive that fall within it.

During the selection process, a hyper-box is chosen first, based on its fitness, and then a random individual within that selected hyper-box is picked. Hyper-boxes with a lower density of individuals are more likely to be selected, favoring regions of the objective space that are less crowded. This approach facilitates diversity in the selection on the approximated Pareto front, as it increases the probability of selecting individuals from less populated regions of the objective space.

This method of selection is advantageous as it simultaneously promotes convergence towards the Pareto front (by selecting hyper-boxes containing non-dominated solutions) and diversity across the Pareto front (by favoring less populated hyper-boxes). Consequently, PESA-II enhances the chances of discovering the true Pareto frontier. According to [4], this hyper-box-based approach is more effective in selecting individuals from less crowded areas compared to the conventional individual-based selection strategy that the original PESA algorithm uses.

A downside to using PESA-II like any other MaOEA, is the fact that it does not scale well with more number of objectives. This is why our contribution includes a modified version of the PESA-II algorithm that can deal with a large number of objectives.

3 APPROACH

To evaluate the performance of PESA-II and DynaPESA-II, we first had to implement the algorithms. This section will firstly explain our adaptation of the PESA-II algorithm for test case generation. Furthermore, we will cover the additional features augmented resulting in DynaPESA-II.

3.1 PESA-II Adaptation

The mutation and crossover operators remain the same in our adaptation as in every abstract EA. Our main adaptation involved modifying the selection process.

Environmental Selection: Instead of selecting the whole Pareto

Front for survival into the next generation, we had to be a bit more selective. As explained before, when we have too many objectives, solutions will be less likely to dominate each other and adding the whole front would not be beneficial. This is why we use the hyper-box classification to help us select diverse solutions. Firstly, the solutions in the Pareto Front are selected and added to a hyper-grid structure. We then iterate over each hyper-box in the hyper-grid, randomly selecting one solution from each hyper-box. The selected solutions proceed to the next generation.
Mating Selection: We assign fitness values to the solutions based on the density of the hyper-box that they belong to. Binary Tournament selection is performed in order to choose the parents, with the solutions from hyper-boxes of lower densities having a higher chance of being selected.
The pseudo-code for this algorithm is given in Algorithm 1.

Algorithm 1: PESA-II

```
Input: $U = \{U_1, ..., U_m\}$: the set of coverage targets of a
        program.
Input: Population size $M$
Result: A test suite $T$
1  $t = 0$                          // current generation
2  $P_t$ = RANDOM-POPULATION($M$)
3  while not search_budget_consumed do
4      $Q_t$ = GENERATE-OFFSPRING($P_t$)
5      $R_t = P_t \cup Q_t$
6      $F$ = GET-PARETO-FRONT($R_t$)
7      $H$ = PESAII-SORTING($F$)  // returns hyper-boxes
        in ascending order of densities
8
9      $P_{t+1} = \emptyset, d = 0$
10     while $d < |H|$ do
11         $P_{t+1} = P_{t+1} \cup$ RANDOMLY-PICK-ONE($H[d]$)
12         $d = d + 1$
13     end
14     $t = t + 1$
15 end
16 $T = P_t$
```

3.2 DynaPESA-II

DynaMOSA introduced features that improve the performance of algorithms for test case generation. We augmented the same features into PESA-II to improve its performance that we will go over.

3.2.1 Preference Criterion: To make sure that test cases that optimize a single test target survive, DynaPESA-II immediately adds them to the next population by adding the Front 0 obtained using the Preference Criterion (used in MOSA). Furthermore, we perform a similar approach as that described in the previous algorithm on front 1 (the rest of the Pareto Front) of the Preference Criterion result.

3.2.2 Archive: DynaPESA-II incorporates the use of the archive in the same way as MOSA does. It adds the test cases which satisfy the previously uncovered branches, and removes those branches from the current set of objectives.

3.2.3 Dynamic Selection of Optimization Targets: DynaPESA-II dynamically adjusts the objectives depending on the structural dependencies. It ignores objectives that are dependent on other uncovered objectives. The pseudo-code for this algorithm is given in Algorithm 2.

Algorithm 2: DynaPESA-II

```
Input: $U = \{U_1, ..., U_m\}$: the set of coverage targets of a
        program.
Input: Population size $M$
Result: A test suite $T$
1  $t = 0$                          // current generation
2  $P_t$ = RANDOM-POPULATION($M$)
3  archive = UPDATE-ARCHIVE($P_t, \emptyset$)
4  while not search_budget_consumed do
5      $Q_t$ = GENERATE-OFFSPRING($P_t$)
6      archive = UPDATE-ARCHIVE($Q_t$, archive)
7      $R_t = P_t \cup Q_t$
8      $F$ = PREFERENCE-SORTING($R_t$)
9      $P_{t+1} = F_0$         // first front is always added
10
11     $H$ = PESA2-SORTING($F_1$)      // according to the
        hyper-box densities
12     $d = 0$
13     while $d < |H|$ do
14         $P_{t+1} = P_{t+1} \cup$ RANDOMLY-PICK-ONE($H[d]$)
15         $d = d + 1$
16     end
17     $t = t + 1$
18 end
19 $T$ = archive
```

4 STUDY DESIGN

This section will provide the details of the experiment that is used to evaluate our approach. Firstly, we precisely explain the research questions that we aim to answer. Following that, we will discuss the benchmark used for the evaluation. After that, we will give an overview of the implementation of the algorithm (in the Syntest framework) and cover the parameters used in the experiments along with the chosen values. Furthermore, the experimental protocol used to compare our approach with the previous algorithms will be explained.

4.1 Research Questions

Firstly, we want to find out whether augmenting DynaMOSA features to the base version of our algorithm results in an improvement in the performance of the algorithm. Hence, we will answer the following research question:

RQ1: How does the base version of PESA-II perform in generating test cases for JavaScript programs compared to DynaPESA-II on branch coverage?
Next, we will find out whether our approach outperforms the current state of the art- DynaMOSA. Hence, we have formulated the following research question:
RQ2: How does DynaPESA-II perform in generating test cases for JavaScript programs compared to DynaMOSA on branch coverage?
These two research questions will be answered in Section 5 by performing statistical analysis of the results of the experiments.

4.2 Benchmark

SynTest JavaScript Benchmark [11] is a tool created to test the performance of the SynTest framework in generating test cases for JavaScript programs. It consists of five JavaScript projects:

- Express[1]
- Commander.js[2]
- Moment.js[3]
- JavaScript Algorithms[4]
- Lodash[5]

The projects chosen for the study were picked for their popularity among the JavaScript community, as indicated by GitHub stars, and for showcasing varied JavaScript syntax and code styles. From these, specific units (such as classes or functions) were selected if they were exportable (testable) and had a Cyclomatic Complexity of at least 2, aligning with established standards for evaluating test case generation tools. The repository is provided by the URL in [10].

We had to exclude the Moment project and the application.js file from the Express project from the experiments since they were causing errors while we ran the benchmark on them. Due to the time constraints of this project, we could not try to fix these errors, hence, we chose to exclude the files.

4.3 Configurations

In order to assess our approach, we created implementations of the algorithms (PESA-II and DynaPESA-II) integrated into the SynTest framework, written in TypeScript. These implementations can be used for unit-level test case generation of JavaScript programs and produce a test suite for the programs under test.

In order to address our research questions, we ran the benchmark using six different configurations for comparison purposes. These configurations encompass both the base algorithms - specifically, NSGA-II, MOSA, and DynaMOSA - as well as our own implementations, namely PESA-II, DynaPESA-II with the "uncovered" objective manager (which disregards the set of covered objectives), and DynaPESA-II with the "structural-uncovered" objective manager (which, in addition to the "uncovered" objective manager, excludes objectives with structural dependencies on any other uncovered objective).

[1]https://expressjs.com/
[2]https://tj.github.io/commander.js/
[3]https://momentjs.com/
[4]https://github.com/trekhleb/javascript-algorithms
[5]https://lodash.com/

4.4 Parameters

The important parameters are listed along with their values and the reason we chose these values. These are the values used for all configurations unless mentioned otherwise.
Population Size: We used the default value in SynTest - 50. PESA-II does not clearly mention a fixed population size to be used which is why we chose the SynTest default.
Hyper-grid Size: 3^n where n is the number of objectives. The PESA-II paper uses a $32x32$ grid size. We could not choose this value since we would be dealing with more than 2 objectives. Another option was to use 32^n grid size but this would not make sense because it would be very rare for multiple solutions to share the same hyper-box. Our aim to create diversity would not be achieved in this case.
Crossover: The crossover probablity is set to 0.7. This is the default value of the crossover operator in SynTest and the same value is used in the generic PESA-II [4].
Mutation: We use the default mutation operator of SynTest where the mutation rate is set to $1/size$ where size is the "number of statements in the test case to mutate" [9]
Search Time: 60 seconds (per run).
Some of the parameters used could have been optimized, however due to the time constraints of this project and the time taken for a single run of the benchmark (approx. 45 minutes) it was not feasible to optimize the parameters.

4.5 Experimental Protocol

To obtain the results for the performance of the aforementioned configurations, we need to run the benchmark on each of the configurations. Since the time taken to converge to the optimal solution impacts the performance, all the algorithms must be run in the same environment. The details of the system used for running the configurations are summarized in table 1.

Component	Specifications
Processor Model	AMD EPYC 7H12
Number of Processors	2
Cores per Processor	64
Total Cores	128
Total Threads	256
Clock Speed	3293.082 MHz
Memory	512GB

Table 1: Summary of Computer System Specifications

The algorithms that are being tested are stochastic in nature. This means that different runs can lead to different results due randomness. To address this issue, the benchmark was run 10 times for each configuration. To measure the final coverage and coverage over time, we took the average of the runs. In order to assess the relative performance of different approaches, we utilized the unpaired Wilcoxon signed-rank test [3] with a significance level of 0.05. This statistical test, which does not rely on assumptions about the data distribution, allows us to determine whether two sets of data are significantly distinct. Moreover, we employed the Vargha-Delaney A_{12} statistic [13] to measure the effect size of the findings, providing insight into the magnitude of the disparity between the two data distributions.

5 RESULTS

This section will present and analyse the results of the experiments conducted with the aim of answering the research questions formulated in Section 4. Tables 2 and 3 summarize the results corresponding to research questions 1 and 2 respectively. The first column shows the class under test, while the rest of the columns display the branch coverage of each algorithm, followed by the statistical tests on the coverage data. The algorithm that obtains a higher branch coverage is highlighted in green for each class.

Some of the files have been excluded from the tables because they had 0% branch coverage for every algorithm. These results would not be beneficial in comparing the performance of the algorithms. The files that have been excluded for this reason are: *articulationPoints.js*, *bellmanFord.js*, *bfTravellingSalesman.js*, *depthFirstSearch.js*, *detectDirectedCycle.js*, *detectUndirectedCycle.js*, *eulerianPath.js*, *floydWarshall.js*, *hamiltonianCycle.js* and *stronglyConnectedComponents.js*.

5.1 Results for RQ-1:

Table 2 gives a summary of the comparison between PESA-II and DynaPESA-II. Overall, DynaPESA-II achieves superior branch coverage results. It outperforms PESA-II in terms of branch coverage for 15 classes. Both achieve the same results for 10 classes, and PESA-II performs better for 1 class. The A_{12} statistic shows a significant difference (large) for the classes where DynaPESA-II outperforms PESA-II. This gives us conclusive evidence that DynaPESA-II is the superior algorithm.

5.2 Results for RQ-2:

Table 3 gives a summary of the comparison between DynaMOSA and DynaPESA-II. Overall, both algorithms achieve similar results, however, DynaMOSA performs slightly better in certain instances. It achieves a higher branch coverage in 6 classes and a lower branch coverage in only 1 class (although the difference is negligible). In the rest of the 20 classes, both algorithms achieve the same branch coverage. Only 3 classes show a large difference in favor of DynaMOSA according to the A_{12} estimate. This leads us to believe that DynaMOSA performs slightly better than DynaPESA-II.

6 THREATS TO VALIDITY

External Validity: One potential threat to the validity of our study is the generalizability of our findings. We selected five open-source projects based on their popularity within the JavaScript community, aiming to capture diversity in terms of size, application domain, purpose, syntax, and code style. However, conducting further experiments on a larger set of projects would enhance the confidence in the generalizability of our study. Therefore, investigating a broader range of projects is part of our future work.

Conclusion Validity: Threats to conclusion validity are associated with the randomized nature of DynaMOSA. To mitigate this risk, we executed each configuration ten times. Following well-established guidelines, we adhered to best practices for running experiments with randomized algorithms. Moreover, we employed the unpaired Wilcoxon signed-rank test and the Vargha-Delaney $\widehat{A}_{12}$ effect size to assess the significance and magnitude of our results. To ensure a controlled environment that facilitates fair evaluation, all experiments were conducted on the same system, with minimal interference from other processes.

7 RESPONSIBLE RESEARCH

Responsible research is important to ensure the ethical and reliable advancement of scientific knowledge. In this study, we adhere to the principles of responsible research to maintain the integrity and validity of our findings.

Firstly, we prioritize transparency and reproducibility by providing an open-access replication package[6]. This package contains the necessary implementations of the PESA-II algorithm adapted to test case generation and the augmented version, DynaPESA-II. Alongside the implementations, we include the method of regenerating the results and the scripts for statistical analysis. By making these resources openly available, other researchers can replicate our experiments, validate our findings, and build upon our work.

Furthermore, we have incorporated appropriate evaluation metrics to assess the quality of the generated test cases. Specifically, we measure branch and method coverage to gauge the effectiveness of the algorithms. These metrics are widely recognized in the software testing community and provide meaningful insights into the thoroughness of the test suite. By employing established evaluation measures, we ensure that our research aligns with standard practices and facilitates comparisons with related studies.

Ethical considerations also play a crucial role in responsible research. We have conducted our experiments within a controlled environment to ensure fair evaluation. All experiments were performed on the same system to maintain consistency. Additionally, our research involves the analysis of open-source JavaScript projects, which have been used for similar evaluations in previous literature. Only one file belonging to the Express project was excluded from testing because it started a web server to generate test cases which caused problems while running the benchmark.

8 CONCLUSION AND FUTURE WORK

In conclusion, this research aimed to adapt PESA-II for test case generation and compare its performance to the current leading algorithm, DynaMOSA. We introduced a base implementation of PESA-II and an enhanced version, DynaPESA-II, incorporating DynaMOSA features. Comparing the results of DynaPESA-II, PESA-II, and DynaMOSA, we observed a significant improvement in DynaPESA-II over PESA-II; however, DynaMOSA remained the superior algorithm. Thus, DynaMOSA remains the preferred choice for generating test cases for JavaScript programs.

Given the limited resources available in this project (time and computation), we utilized default parameters for the experiment. Future investigations could explore the potential enhancements achieved by parameter tuning. Additionally, our adaptation of PESA-II for test case generation represents just one of several possible approaches. Exploring alternative strategies could offer valuable insights and yield different outcomes. These considerations should be considered in future research endeavors within this domain.

[6]https://github.com/Abhishek-A-dev/CSE_3000_Replication

Table 2: Statistics comparing DynaPESA-II and PESA-II on branch coverage

Classes	PESA-II	DynaPESA-II	p_values	a12	a12_estimate
help.js	32.6%	50.0%	1.09×10^{-4}	1.00	large
option.js	44.4%	44.4%	0.32	0.63	small
suggestSimilar.js	65.6%	68.8%	0.05	0.76	large
query.js	33.3%	66.7%	1.59×10^{-5}	1.00	large
request.js	23.9%	32.6%	4.88×10^{-5}	1.00	large
response.js	16.3%	17.4%	3.87×10^{-3}	0.88	large
utils.js	26.1%	43.5%	1.32×10^{-4}	1.00	large
view.js	37.5%	37.5%	NA	0.50	negligible
breadthFirstSearch.js	25.0%	12.5%	0.03	0.25	large
kruskal.js	20.0%	20.0%	NA	0.50	negligible
prim.js	16.7%	16.7%	NA	0.50	negligible
Knapsack.js	42.5%	57.5%	5.47×10^{-5}	1.00	large
KnapsackItem.js	50.0%	50.0%	NA	0.50	negligible
Matrix.js	7.9%	7.9%	NA	0.50	negligible
CountingSort.js	57.1%	57.1%	0.58	0.55	negligible
RedBlackTree.js	29.4%	29.4%	0.37	0.55	negligible
equalArrays.js	72.9%	75.0%	0.44	0.61	small
hasPath.js	68.8%	100.0%	5.11×10^{-5}	1.00	large
random.js	57.1%	100.0%	9.98×10^{-5}	1.00	large
result.js	80.0%	80.0%	0.30	0.60	small
slice.js	90.0%	100.0%	1.99×10^{-4}	0.95	large
split.js	87.5%	87.5%	0.03	0.70	medium
toNumber.js	50.0%	65.0%	2.15×10^{-4}	0.95	large
transform.js	58.3%	75.0%	0.01	0.86	large
truncate.js	52.9%	55.9%	3.32×10^{-5}	1.00	large
unzip.js	16.7%	100.0%	2.04×10^{-4}	0.97	large

Table 3: Statistics comparing DynaMOSA and DynaPESA-II on branch coverage

Classes	DynaMOSA (%)	DynaPESA-II (%)	p_values	a12	a12_estimate
help.js	50.00	50.00	0.17	0.40	small
option.js	50.00	44.44	0.03	0.23	large
suggestSimilar.js	71.88	68.75	0.06	0.28	medium
query.js	66.67	66.67	NA	0.50	negligible
request.js	32.61	32.61	0.37	0.55	negligible
response.js	19.57	17.39	0.00	0.08	large
utils.js	42.39	43.48	0.69	0.55	negligible
view.js	37.50	37.50	0.37	0.45	negligible
breadthFirstSearch.js	18.75	12.50	0.69	0.45	negligible
kruskal.js	20.00	20.00	NA	0.50	negligible
prim.js	16.67	16.67	NA	0.50	negligible
Knapsack.js	57.50	57.50	NA	0.50	negligible
KnapsackItem.js	50.00	50.00	NA	0.50	negligible
Matrix.js	7.89	7.89	NA	0.50	negligible
CountingSort.js	57.14	57.14	1.00	0.50	negligible
RedBlackTree.js	29.41	29.41	NA	0.50	negligible
equalArrays.js	83.33	75.00	0.01	0.19	large
hasPath.js	100.00	100.00	NA	0.50	negligible
random.js	100.00	100.00	0.17	0.40	small
result.js	80.00	80.00	0.17	0.40	small
slice.js	100.00	100.00	NA	0.50	negligible
split.js	87.50	87.50	NA	0.50	negligible
toNumber.js	65.00	65.00	NA	0.50	negligible
transform.js	91.67	75.00	0.09	0.29	medium
truncate.js	55.88	55.88	0.17	0.40	small
unzip.js	100.00	100.00	0.58	0.45	negligible

REFERENCES

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001. https://doi.org/10.1016/j.jss.2013.02.061

[2] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. 2014. Evolutionary Algorithms. *WIREs Data Mining and Knowledge Discovery* 4, 3 (2014), 178–195. https://doi.org/10.1002/widm.1124 arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1124

[3] W.J. Conniver. 1998. *Practical Nonparametric Statistics*. Vol. 350. Wiley, Hoboken.

[4] 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.

[5] 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.

[6] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, Open Problems and Challenges for Search Based Software Testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–12. https://doi.org/10.1109/ICST.2015.7102580

[7] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. 2015. Many-Objective Evolutionary Algorithms: A Survey. *ACM Comput. Surv.* 48, 1, Article 13 (sep 2015), 35 pages. https://doi.org/10.1145/2792984

[8] 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

[9] 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

[10] Dimitri Stallenberg and Mitchell Olsthoorn. 2021. *Syntest JavaScript Benchmark*. https://github.com/syntest-framework/syntest-javascript-benchmark

[11] 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.

[12] Tea Tušar and Bogdan Filipič. 2015. Visualization of Pareto Front Approximations in Evolutionary Multiobjective Optimization: A Critical Review and the Prosection Method. *IEEE Transactions on Evolutionary Computation* 19, 2 (2015), 225–245. https://doi.org/10.1109/TEVC.2014.2313407

[13] 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