Augmenting Pareto Corner Search Evolutionary Algorithm for Automatic Test Case Generation

Emin Alp Guneri[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: Emin Alp Guneri
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Dimitri Stallenberg, Sicco Verwer

ABSTRACT

Software testing is a laborious job, and accounts for a large portion of software development expenses. Search-based automatic test case generation is an area of research that attempts to remedy this by discovering algorithms suited for generating test cases automatically. In this field, DynaMOSA is a state-of-the-art evolutionary algorithm, which reduces the problem of test case generation to a multi-objective optimization problem, and uses domain knowledge to generate solutions efficiently. In this paper, we adapt Pareto Corner Search Evolutionary Algorithm (PCSEA) for test case generation. Furthermore, we integrate DynaMOSA heuristics into PCSEA, and create a novel algorithm, DynaMOSAPCSEA. We evaluate the test case generation efficacy of PCSEA, DynaMOSA, and DynaMOSAPCSEA by using the JavaScript benchmark provided by SynTest Framework. The results indicate that PCSEA is a feasible algorithm for test case generation, however DynaMOSA heuristics improve its performance only minimally.

1 INTRODUCTION

Software testing is the process of verifying that the software under development behaves as expected. Alongside verifying that the product is working properly, software testing also improves the quality of the product (by enforcing good design practices), and increases customer trust and satisfaction [16]. For these reasons, software testing is a crucial component of the development process.

In practice, software testing is usually done by developers. This involves a lot of manual labor, especially for complicated systems with a high number of branches. Anand et al. claim that software testing is a major contributor to development costs, and using automated tools to generate test cases is a promising solution; automated tools can improve product quality while reducing development costs[1].

Automated tools for generating test cases are a popular topic in research. There are a variety of techniques proposed in literature; techniques range from symbolic execution to adaptive random testing[1]. As of today, there are numerous automated testing tools available in the market. In fact, IT companies are beginning to make use of these tools to augment their test suites[1].

A prominent approach to the problem of automatic test case generation is search-based testing. This approach uses a search algorithm to efficiently navigate the search space and find good test cases for the system under test. A common search-based approach is to translate covering a statement/branch to minimizing a function, which is called an objective. Panichella et al. provide a formal formulation of automatic test case generation as a multi-objective optimization problem[13]. In return, generating a good test suite generally becomes a problem of finding a set of solutions that minimize a large number of objectives.

Evolutionary algorithms are a natural choice for multi-objective optimization problems, since they are simple to understand and implement, and do not require additional information[4]. NSGA-II has been the most popular evolutionary algorithm for solving multi-objective optimization problems; it has acted as a benchmark to assess if new multi-objective algorithms were feasible[8]. Due to its widespread influence, there have been many variants of NSGA-II proposed in the literature, such as R-NSGA-II[6] and NSGA-III[10].

There are many more evolutionary algorithms in literature that provide promising performances, however, all these algorithms are intended to work with numerical problems with a low number of objectives. Multi-objective optimization problems that have more than 3 objectives are called many-objective optimization problems (MaOP)[14], and they occur commonly in the context of test case generation. Evolutionary algorithms such as NSGA-II do not scale well for such high numbers of objectives [9].

The state-of-the-art evolutionary algorithms each have their own nuances to excel under different circumstances. However, almost all of them struggle with MaOPs, because 1- the ratio of non-dominated solutions in the population increase with number of objectives, making it difficult to evolve the population towards optimal solutions, and 2- the Pareto front that the algorithms try to approximate grows exponentially larger with increasing number of objectives[14]. In the problem of test case generation, because each statement or branch presents an objective, using a evolutionary multi-objective algorithm to search the entire solution space is totally unfeasible for complex programs.

Dynamic Multi-Objective Sorting Algorithm (DynaMOSA) is an algorithm that is specifically suited for solving MaOPs generated by test case generation problems[13]. DynaMOSA improves upon NSGA-II using domain knowledge. Instead of approximating the entire Pareto front, DynaMOSA aims to evolve the population towards the boundaries of the Pareto front, and capture the solutions that reach these boundaries in an archive. The solutions that lie on the boundaries of the Pareto front (i.e. solutions that minimize one or more objectives) are remarkable in context of test case generation, since they represent test cases that cover one or more testing targets (branches/statements/etc.). These solutions are also called Pareto corner solutions. With enough such solutions, it might be possible to exercise all the testing targets of the system under test.

One thing to note is that, Pareto corners are also quite useful for other purposes (e.g. dimensionality reduction)[14]. Thus, there are also algorithms geared towards finding corner solutions in different domains. For example, Pareto Corner Search Evolutionary Algorithm (PCSEA) is an algorithm that approximates Pareto corner solutions by using a ranking strategy to rank solutions near the corners of the Pareto front highly. This makes PCSEA also a good candidate for automatic test case generation. This raises the question of whether PCSEA could act as a better baseline algorithm than NSGA-II for DynaMOSA.

To this end, we implement PCSEA in SynTest, so that it can be used to generate tests for real JavaScript programs. We also make use of DynaMOSA heuristics to create an improved version of PCSEA which we call DynaMOSAPCSEA. We run NSGA-II, PCSEA, DynaMOSA, and DynaMOSAPCSEA on popular npm packages to assess their test generation efficacy, and conduct statistical analysis to compare the branch coverages they achieve.

This paper contributes to the current literature as follows.

- We adapt PCSEA to be able to generate test cases.
- We improve PCSEA with DynaMOSA heuristics.

- We measure the branch coverages achieved by our PCSEA and DynaMOSAPCSEA implementations, and analyze their performance.
- We provide a replication package[1] to allow the readers to conduct the experiments.

This paper is organized as follows: Section 2 provides a formal description of the automatic test case generation. Following this, Section 3 introduces the algorithms that were compared on how effective they can generate tests for JavaScript programs. Section 4 goes into detail about the experiments that were run to compare the algorithms. The results of these experiments are provided in Section 5, and are analyzed in Section 6. The threats to the validity of these results and findings are addressed in Section 7. Section 8 is dedicated to responsible research, and considers the ethical and societal implications of the collected results, alongside verifying that the conducted experiments are reproducible. Finally, Section 9 concludes the paper, summarizing the important results, and lays out future work suggestions.

2 BACKGROUND

Multi-objective optimization problems occur commonly in multiple engineering fields, thus there have been countless methods proposed to solve them [11]. These optimization problems usually stem from physical problems, and thus involve minimizing/maximizing a set of numerical objective functions that may or may not conflict with other objectives.

In the context of test case generation, each branch/statement presents an objective. The objective function returns, for a given test case, how close the test case is to covering the branch/statement. An objective function only returns non-negative values, with 0 indicating that the test case covers the branch/statement, and a large value indicating that the test case must change significantly to satisfy the predicates necessary to cover the branch/statement. In real life software, we usually observe that the branches stemming from the same conditional expression can not be covered at the same time. The objectives of these branches are completely conflicting objectives. On the other hand, covering a branch might rely on following a certain sequence of branches. In this case, the objectives of the branches in the sequence are redundant, because covering the dependent branch already requires covering all the previous branches in the sequence.

For problems that involve conflicting objectives, such as test case generation, the concept of Pareto dominance is crucial. A solution X is said to dominate another solution Y, if it performs better on at least one objective, and the same on the other objectives. In short, it could be thought as X being strictly better than Y.

Pareto dominance relation partitions the population into sets that are called fronts. First front F_1 contains individuals in the population that are not dominated by any other individual in the population. Second front F_2 contains individuals that are only dominated by individuals in F_1, and so on. It should be noted that individuals in the same front do not dominate each other, so none of them could be considered better than the others.

[1] https://github.com/Alp-Guneri/SynTest-Replication-Package.git

Another crucial concept is Pareto optimality. A solution is Pareto optimal if there does not exist a single possible solution that dominates it[3]. The set of all Pareto optimal solutions is called a Pareto front, and the aim of multi-objective optimization problems is to determine an approximate subset of the Pareto front.

There are numerous ways to approach such multi-objective optimization problems. Metaheuristic search algorithms, such as hill climbing, simulated annealing, and evolutionary algorithms are intuitive, and easy to implement, making them a common choice for such problems. In this paper, we only make use of evolutionary algorithms.

Evolutionary algorithms use encodings; each individual in the population is a string, binary number, or a scheme of representing a potential solution to the problem. Once the encoding is decided, evolutionary algorithms begin with a population of random individuals. In each generation, these individuals generate offsprings (new individuals). This is done by a selection operator which chooses parents from the current population, and crossover operator which describes how the offspring should be created from the information of the parents. Then, the population is updated using these offsprings.

The population size is usually fixed, meaning some of the solutions must be abandoned. This process is called environmental selection, and the good candidates in the population are selected to survive to the next generation. The survivors have a probability to mutate, and these mutations are described by the mutation operator. Competent evolutionary algorithms aim to promote diversity (so that a lot of different possible solutions are explored) through mutation and crossover, along with optimality through environmental and parental selection.

Each population serves as an intermediary approximation of the Pareto front, that gets more accurate per generation. The final population serves as the algorithm's approximation of the Pareto front.

To make this paper relatively self-contained, this section will briefly explain the algorithms that will be used in this paper. We first introduce NSGA-II and PCSEA, two evolutionary algorithms for solving multi-objective optimization problems. Then, we introduce DynaMOSA, an evolutionary algorithm which uses domain knowledge to be more efficient at test case generation.

2.1 NSGA-II

NSGA-II begins with a random population. The parental selection is done by binary tournament, and the selected parents are crossed over with a probabaility p_c to produce an offspring. After offspring population is obtained, the parent and offspring population are joined.

The joint population is partitioned into Pareto fronts based on Pareto dominance, and the individuals in the fronts $F_1, F_2 ... F_m$ are added to the next generation until F_{m+1} contains more individuals than necessary. The individuals in F_{m+1} are sorted on their crowding distance, which is a density estimator that tells how dense the individual's region is. If there are many individuals similar to the given individual, this individual contributes minimally to the diversity of the population, and has a low crowding distance. Only the solutions with the highest crowding distances in F_{m+1} are chosen

for the next population, which acts as a secondary mechanism to promote diversity in the population.

The population is then mutated, with a mutation probability p_m. The authors of NSGA-II propose setting $p_m = \frac{1}{\#ofobjectives}$[5].

2.2 PCSEA

Pareto Corner Search Evolutionary Algorithm (PCSEA) is an evolutionary algorithm that finds corner solutions to eliminate redundant objectives from the problem[14]. Once the number of objectives is reduced, an evolutionary algorithm such as NSGA-II could be used to solve the reduced numerical problem.

Finding Corner Solutions

A set of corner solutions are calculated in the first half of the algorithm. This part of the algorithm is quite similar to NSGA-II. PCSEA makes use of the same selection, crossover, and mutation operators as NSGA-II; it only differs in environmental selection. A procedure called Corner-Sort is used to rank the solutions in the joint parent-offspring population. Corner sort follows the following steps[14].

(1) The solutions are sorted based on each objective, in ascending order. For ith objective f_i, a solution x should appear before solution y, if and only if $f_i(x) < f_i(y)$.
(2) The solutions are sorted based on the all the objectives except the current one. For ith objective f_i a solution x should appear before solution y if and only if $\sum_{j=1,j\neq i}^{j=M} f_j(x)^2 < \sum_{j=1,j\neq i}^{j=M} f_j(y)^2$.

This procedure gives a total of $2M$ sorted lists. The first solution in the first sorted list is assigned rank 1. The first solution in the second list is assigned rank 2. This goes on until the first solution in each sorted list is assigned a rank. If a solution that was already ranked gets chosen again, the next element in the list is chosen instead. The individuals in the joint population with the highest ranks are chosen, and the individuals with low ranks are abandoned.

Removing Redundant Objectives

In the second half of the algorithm, these corner solutions are used to determine if an objective is redundant, i.e. the removal of the objective preserves the dominance relations between solutions. The impact of removing the objective entirely is quantified by the ratio of non-dominated solutions before and after removing an objective[14]. If this ratio is above a chosen threshold, the objective is deemed redundant, and is removed. This process is repeated sequentially for each objective to remove the redundant ones. The final result is a smaller set of objectives that serves as a representative of the initial problem.

2.3 DynaMOSA

DynaMOSA is a multi-objective evolutionary algorithm, geared towards test case generation[13]. Instead of attempting to approximate the entire Pareto front like NSGA-II, DynaMOSA attempts to foster a population that would make a good test suite.

DynaMOSA starts off similar to NSGA-II, however introduces three new heuristics.

- **Dynamic objectives.** In test case generation, some testing targets are directly dependent on others being covered. This means we should consider satisfying the controlling target first, and then try covering the dependent target. This allows us to keep a smaller and dynamic set of objectives while building a test suite. Initially, the only objective is the root statement/branch of the method under test. Once a target gets covered, the objective functions of the dependent testing targets are appended, and the objective function of the covered target is removed. This can significantly reduce the number of objectives in a long method with little nesting.
- **Preference sorting.** This method is quite similar to non-dominated sorting in NSGA-II, however prioritizes solutions that minimize at least one objective. Solutions that minimize at least one objective the most in the current population are assigned to F_0, and the remaining solutions are sorted by fast-nondominated-sort. This way, a preference towards the boundaries of the Pareto front is introduced, and the population should evolve more individuals that completely minimize at least one objective.
- **Archiving.** Archiving is not unique to the field of test case generation. Archiving is usually done to make sure that the Pareto optimal solutions that emerge in a population do not die out in later generations[12]. In DynaMOSA, an archive is used to preserve test cases that cover a previously uncovered testing target. An important thing to note is that the saved test case might be replaced by another test case, if both of them cover the same target, but the new one is shorter than the previous one. This attempts to make the final test suite concise and human-friendly.

One final thing to note is that DynaMOSA returns the archive that it repeatedly updates throughout generations, unlike NSGA-II which returns the final population. This is a more reasonable approach considering that DynaMOSA relies on certain targets being covered. If the solutions that cover these targets are not present in the final solution, the algorithm will have a lower coverage. Archiving preserves these test cases, and makes sure the test suite contains at least one test case that covers the testing targets that were marked as covered.

3 APPROACH

This section describes how PCSEA was adapted to the problem of test case generation, and how it was improved with DynaMOSA heuristics.

3.1 PCSEA for Test Case Generation

PCSEA is an algorithm suited for numerical problems. However, it could work well on test case generation, since the first half of the algorithm, introduced in Section 2.2, attempts to find corner solutions (i.e. solutions that completely minimize one or more objectives). The corner solutions in test case generation would be solutions that achieve a value of 0 (since this is the lowest value an objective function could return) for one or more objectives, meaning they would be test cases that cover one or more branches/statements. Thus, we use only the first half of the algorithm.

PCSEA requires an encoding scheme before it is able to generate test cases. One way to encode test cases is to consider them as sequences of statements $t = \langle s_1, s_2, ..., s_l \rangle$[7]. Each statement could be a (1) Primitive statement, (2) Constructor statement, (3) Field statement, (4) Method statement, or (5) Assignment statement. This encoding captures the way humans usually test systems in a unit testing level, allowing PCSEA to be used for generating unit tests.

DynaMOSAPCSEA

PCSEA appears to be a good candidate for test case generation. However, we conjecture that its performance can be improved by integrating DynaMOSA heuristics into PCSEA, to make it more suited for generating test cases.

We integrate the three DynaMOSA heuristics listed in Section 2.3. Dynamic objectives heuristic allows DynaMOSAPCSEA to work on a smaller dynamic set of objectives. Archiving heuristic allows DynaMOSAPCSEA to save important test cases to its test suite. Preference sorting heuristic, on the other hand, is something already done by Corner-Sort procedure. Both sorting methods rank the individuals in the population that minimize an objective as the best ones. We integrate preference sorting into PCSEA by running preference sorting first, and appending the individuals in F_0 to the next population. Then, we run Corner-Sort to rank the remaining population.

We end up with an algorithm that resembles DynaMOSA quite a bit. DynaMOSAPCSEA differs from DynaMOSA in how it handles environmental selection. Instead of the non-dominated sorting and crowding distance assignment, the environmental selection is handled by the Corner-Sort method.

4 STUDY DESIGN

This sections aims to explain the details about the experiments that we carried out. The experiments we conducted can be replicated with the replication package[2].

4.1 Research Questions

The aim of this study is to assess whether DynaMOSA heuristics apply well to PCSEA. To this end, we have formulated the following research question.

- **RQ1:** How do PCSEA and DynaMOSAPCSEA perform compared to each other with regards to branch coverage?
- **RQ2:** How does DynaMOSAPCSEA perform compared to DynaMOSA with regards to branch coverage?

With the first research question, we aim to determine if DynaMOSA heuristics improve the performance of PCSEA for test case generation. With the second research question, we aim to determine if DynaMOSAPCSEA is a feasible algorithm for test case generation, or if it is overshadowed by the state-of-the-art algorithm, DynaMOSA.

4.2 Configurations

The research questions in Section 4.1 involve comparing algorithms. To make a fair judgement in our comparisons, we will run the algorithms, and conduct statistical analysis to determine whether an algorithm achieves a significantly better result than the other. The results of the following pairs of algorithms will be compared with statistical analysis.

- DynaMOSAPCSEA vs PCSEA
- DynaMOSAPCSEA vs DynaMOSA

4.3 Implementation

SynTest[3] is a tool created for testing JavaScript programs. At the heart of this tool is the syntest-core[4] repository, which provides numerous necessary utility functions, alongside the plugins necessary to run search algorithms. These utility functions handle the logic necessary to read through the source code of the system under test, identify the branches, and reduce the problem to a many-objective optimization problem.

The implementation of PCSEA and DynaMOSAPCSEA were done by creating a fork[5] of syntest-core repository, implementing these evolutionary algorithms, and creating a plugin and preset for each of them. This allows one to use PCSEA and DynaMOSAPCSEA in SynTest's command-line interface.

4.4 Benchmark

SynTest also provides a benchmark to evaluate the performances of the search algorithms. The syntest-javascript-benchmark[6] repository provides the necessary files and instructions for the evaluation process. This benchmark contains hand-picked files from the following popular npm packages.

(1) Commander.js
(2) Express
(3) JavaScript Algorithms
(4) Moment.js
(5) Lodash

The benchmark allows one to run a search algorithm on the benchmark files. The search algorithm generates test cases for the given files, and once the search budget is consumed, the achieved branch and statement coverages are displayed.

We have excluded the files of Moment.js entirely, because they caused the tool to crash. We have also excluded a file from Express, namely application.js, because it causes the tool to get stuck on post-processing, since the generated test cases start up a web server.

4.5 Parameters

There are numerous parameters in the SynTest tool, however, for the most part, we used the default parameter values. The parameters that may affect the result are mainly about crossover, mutation, and procreation. We proceed with default parameters, because it has been demonstrated that using default parameters suggested in the literature for these operators give reasonable performances[2]. Table 1 shows some of these parameters, together with their default values.

Since we want to evaluate the performances of the algorithms fairly, we run the algorithms on the same machine. Table 2 shows

[2]https://github.com/Alp-Guneri/SynTest-Replication-Package.git

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

Parameter	Value
Population Size	50
Search Time	90 seconds
Crossover Probability	0.7
Multi-point Crossover Probability	0.5
Mutation Rate	$\frac{1}{\#objectives}$

Table 1: Default parameter values used in each experiment.

the hardware specifications of the machine that was used to conduct the experiments.

Part	Name	Information
CPU	2x AMD EPYC 7H12 64-Core Processor	3293.082 MHz 128 Cores, 256 Threads
RAM	-	512 GB

Table 2: Hardware specifications of the machine used to run the experiments.

4.6 Experimental Protocol

We treat running one algorithm for one benchmark file as one experiment. For each experiment, we set the time budget to 120 seconds.

We run PCSEA, DynaMOSA, and DynaMOSAPCSEA on all 36 benchmark files, 10 times each. This is done to account for the stochasticity of the algorithms. This gives a total of 3 (algorithms) * 36 (benchmark files) * 10 (repetitions) * 120 (search budget) = 36 hours of sequential run time. To reduce this run time, we used a Python script to conduct 100 experiments in parallel, resulting in a final run time of ≈ 22 minutes.

The results of running the algorithms were saved in separate folders (3 * 36 * 10 = 1080 in total). These folders in return contain files that measure the coverages and other metrics of each experiment over time. The final values of all the metrics each experiment achieved were saved in a csv file at the base folder, for convenience.

We made a Python script to process these results and generate visuals. We used this script to prepare the table of results provided in Section 5, however the script allows for smaller tables to be generated.

Following this, we used an R script to handle statistical analysis of the results. For statistical analysis, we compared the algorithms in pairs listed in Section 4.2, and used p and A-12 values to detect if the differences in the branch coverages they achieved were statistically significant.

5 RESULTS

The algorithms were run, as described in Section 4.6. Listing 1, shows an example test case generated by DynaMOSAPCSEA for help.js file from the Commander.js project. Table 3 displays the branch coverages achieved by each algorithm on the benchmark files, together with comparison statistics. Please note that Table 3 does not include 10 files from JS Algorithms project, because all 3 algorithms that we consider achieved 0% branch coverage on them. This section aims to use the coverage data and statistics we collected to answer the research questions formulated in Section 4.1.

We statistically analyze if DynaMOSAPCSEA provides any advantages by calculating the p-value of the null hypotheses "DynaMOSAPCSEA performs no different than PCSEA for the given benchmark file", and "DynaMOSAPCSEA performs no different than DynaMOSA for the given benchmark file". We consider a p-value lower than 0.005 as a threshold for rejecting the null hypothesis. Furthermore, we calculate the effect size ($\hat{A}_{12}$) for each file, to determine which algorithm performs better, and how much advantage it offers over the other. An $\hat{A}_{12}$ value of 0.5 means identical performance. An $\hat{A}_{12}$ value greater than 0.5 means DynaMOSAPCSEA performs better, and an $\hat{A}_{12}$ value smaller than 0.5 means the other algorithm is better. We summarize the results in Table 4.

Listing 1: An example test case generated by DynaMOSAPCSEA for help.js file from the Commander.js project.

```
describe('help', () => {
  const {Help} = require("../../benchmark/
      ↪ commanderjs/lib/help.js");

  it('test for help', async () => {
    const _Help_object_tYnC = new Help()
    const _flags_numeric_nXBp =
        ↪ -4.374420013188537;
    const _option_object_axhg = {
      "flags": _flags_numeric_nXBp
    }
    const _optionTerm_function_Rxvw = await
        ↪ _Help_object_tYnC.optionTerm(
        ↪ _option_object_axhg)
    const _description_function_CBdi = () =>
        ↪ { };
    const _cmd_object_stMt = {
      "description":
          ↪ _description_function_CBdi
    }
    const
        ↪ _subcommandDescription_function_pgJy
        ↪ = await _Help_object_tYnC.
        ↪ subcommandDescription(
        ↪ _cmd_object_stMt)
    const _description_function_ctPJ = () =>
        ↪ { };
    const _cmd_object_oSPg = {
      "description":
          ↪ _description_function_ctPJ
    }
    const
        ↪ _subcommandDescription_function_hAHg
        ↪ = await _Help_object_tYnC.
        ↪ subcommandDescription(
        ↪ _cmd_object_oSPg)
});
```

Benchmark	File Name	PCSEA	DynaMOSA	DynaMOSAPCSEA	PCSEA vs DynaMOSAPCSEA		DynaMOSA vs DynaMOSAPCSEA	
					p-value	$\hat{A}_{12}$ Value	p-value	$\hat{A}_{12}$ Value
Commander.js	help.js	40.91%	50.00%	50.00%	0.000126	1.0(large)	0.077583	0.35(small)
	option.js	50.00%	50.00%	47.22%	0.414929	0.4(small)	0.336750	0.375(small)
	suggestSimilar.js	71.88%	71.88%	71.88%	0.582778	0.45(negligible)	1.000000	0.5(negligible)
Express	query.js	66.67%	66.67%	66.67%	N/A	0.5(negligible)	N/A	0.5(negligible)
	request.js	32.61%	32.61%	32.61%	N/A	0.5(negligible)	0.368120	0.55(negligible)
	response.js	17.93%	19.57%	19.84%	0.000457	0.965(large)	0.398127	0.615(small)
	utils.js	41.30%	42.39%	42.39%	0.005741	0.825(large)	1.000000	0.5(negligible)
	view.js	37.50%	37.50%	37.50%	0.167489	0.4(small)	0.368120	0.45(negligible)
JS Algorithms	breadthFirstSearch.js	18.75%	18.75%	18.75%	1.000000	0.5(negligible)	1.000000	0.5(negligible)
	kruskal.js	20.00%	20.00%	20.00%	N/A	0.5(negligible)	N/A	0.5(negligible)
	prim.js	16.67%	16.67%	16.67%	N/A	0.5(negligible)	N/A	0.5(negligible)
	Knapsack.js	57.50%	57.50%	57.50%	N/A	0.5(negligible)	N/A	0.5(negligible)
	KnapsackItem.js	50.00%	50.00%	50.00%	N/A	0.5(negligible)	N/A	0.5(negligible)
	Matrix.js	7.89%	7.89%	7.89%	N/A	0.5(negligible)	N/A	0.5(negligible)
	CountingSort.js	57.14%	57.14%	57.14%	0.300558	0.4(small)	0.300558	0.4(small)
	RedBlackTree.js	29.41%	29.41%	29.41%	N/A	0.5(negligible)	N/A	0.5(negligible)
Lodash	equalArrays.js	83.33%	83.33%	83.33%	0.167489	0.4(small)	0.582778	0.45(negligible)
	hasPath.js	100.00%	100.00%	100.00%	N/A	0.5(negligible)	N/A	0.5(negligible)
	random.js	100.00%	100.00%	100.00%	1.000000	0.5(negligible)	0.167489	0.4(small)
	result.js	80.00%	80.00%	80.00%	0.167489	0.4(small)	0.167489	0.4(small)
	slice.js	100.00%	100.00%	100.00%	N/A	0.5(negligible)	N/A	0.5(negligible)
	split.js	87.50%	87.50%	87.50%	N/A	0.5(negligible)	N/A	0.5(negligible)
	toNumber.js	65.00%	65.00%	65.00%	N/A	0.5(negligible)	N/A	0.5(negligible)
	transform.js	91.67%	91.67%	91.67%	0.034843	0.3(medium)	0.931313	0.485(negligible)
	truncate.js	55.88%	55.88%	55.88%	0.368120	0.55(negligible)	0.582778	0.45(negligible)
	unzip.js	100.00%	100.00%	66.67%	0.001617	0.15(large)	0.008670	0.2(large)

Table 3: Median branch coverages achieved by PCSEA, DynaMOSA and DynaMOSAPCSEA on the benchmark files, followed by statistical comparisons of the average branch coverage. 10 files from JS Algorithms are not included in this table, because no branch coverage was achieved.

Comparisons	# Win				# No Diff				# Lose			
	Negl	Small	Medium	Large	Negl	Small	Medium	Large	Negl	Small	Medium	Large
DynaMOSAPCSEA vs PCSEA	-	-	-	2	26	5	1	1	-	-	-	1
DynaMOSAPCSEA vs DynaMOSA	-	-	-	-	29	6	-	1	-	-	-	-

Table 4: Summary of Comparison Statistics for DynaMOSAPCSEA vs PCSEA and DynaMOSAPCSEA vs DynaMOSA.

RQ1: How do PCSEA and DynaMOSAPCSEA perform compared to each other with regards to branch coverage?

As shown in Table 4, DynaMOSAPCSEA only achieves a significant advantage over PCSEA for only 2 files, namely help.js from Commander.js project, and response.js from Express. PCSEA achieves a significant advantage for only 1 file, namely unzip.js from Lodash. The remaining files in the benchmark we used yielded p-values greater than 0.005. This shows that the DynaMOSA heuristics do not impact the performance of PCSEA, because the improvements we observed were minimal.

RQ2: How does DynaMOSAPCSEA perform compared to DynaMOSA with regards to branch coverage?

As shown in Table 4, there were no benchmark files where DynaMOSAPCSEA and DynaMOSA obtained a statistically significant advantage over the other. This shows that DynaMOSAPCSEA is just as performant as DynaMOSA, and thus is a feasible algorithm for test case generation.

6 RESPONSIBLE RESEARCH

The algorithms studied in this paper do not raise significant ethical concerns, neither do they handle sensitive or personal data. The algorithms that we propose are intended for test case generation, and rely on domain knowledge to be efficient. It is not likely for

these algorithms to be used in different contexts (potentially malicious ones) without modifications. We believe that the algorithms studied in this paper can help software developers augment their test suites and publish more reliable software.

The reader can reproduce the experiments conducted, and use the algorithms for different files if they would like to do so. The SynTest framework, the algorithms that were studied, and the benchmark files that were used conduct our experiments are all open source. This makes our algorithms and experiments transparent and reproducible. We also provide a replication package we provide in Section 4, so that the reader can run the same code we used to obtain our results and plots.

7 THREATS TO VALIDITY

In this section, we consider potential factors that might affect the validity of our results and our analysis.

7.1 Threats to Construct Validity

In our analysis, we relied on branch coverage to compare the algorithms. Branch coverage is a well-established metric, and provides a reasonable assessment of the test case generation efficacy of the algorithms.

7.2 Threats to Internal Validity

The algorithms proposed in this paper were implemented and evaluated within the SynTest Framework. The proposed algorithms were unit tested by a synthesized population, and an example population provided in the original PCSEA paper. The framework also contains unit tests for utility functions, genetic operators, and so on. These unit tests give some confidence that the proposed algorithms and the framework behave as intended, however provide no guarantees.

Another factor to consider is the stochasticity of the algorithms. Each algorithm begins with a somewhat random initial population, which usually impacts the final result of the algorithm. To account for this, we ran the algorithms 10 times per benchmark file, and conducted statistical analysis to determine if the results showed statistical significance.

Finally, it must be noted that genetic operators also have an impact on the performance of evolutionary algorithms. In this paper, the same genetic operators from the SynTest framework were used with default parameter values.

7.3 Threats to External Validity

We used an existing benchmark to assess the performances of algorithms. This benchmark is reliable, because it is relatively diverse in terms of the programming constructs used and coding styles[15]. In the future, extending this benchmark with more JavaScript files might provide more evidence to the generalisation of our results.

8 CONCLUSION AND FUTURE WORK

In this paper, we studied the problem of test case generation as a multi-objective optimization problem, and measured the performances of different evolutionary algorithms. We adapted PCSEA for test case generation, and integrated DynaMOSA heuristics into it to create a novel algorithm, DynaMOSAPCSEA. PCSEA required minimal adaptation for test case generation; the Corner-Sort method proposed by Singh et. al was already very suitable for test case generation. DynaMOSAPCSEA introduced three DynaMOSA features on top of our PCSEA implementation: (i) Dynamic Objectives, (ii) Preference Sorting, and (iii) Archiving.

For our empirical study, we used the JavaScript benchmark available on SynTest. This benchmark contains a diverse set of files from popular npm packages. We have observed that PCSEA, DynaMOSA, and DynaMOSAPCSEA obtain very similar results. From this, we conclude that DynaMOSA heuristics do not create a significant improvement on PCSEA's performance. Furthermore, from these results, we claim that PCSEA and DynaMOSAPCSEA are feasible algorithms for test case generation, since their performances match the performance of state-of-the-art algorithm, DynaMOSA.

There is still much future work to do to automate the process of test case generation completely. Here, we include a few research directions that might be interesting to pursue.

First of all, all the algorithms described in this paper made use of default parameter values for NSGA-II. These parameters give reasonable performances, and thus we believe the comparisons were fair. However, tuning the parameter values could definitely increase the branch coverages each algorithm achieves.

Another factor that impacted the performance of the algorithms stems from functional coverage. Even though the functional coverage each algorithm achieves is not included in results, the Python script included in the replication package generates this data as well. A common problem that we noted from inspecting the data we obtained was that the branch coverage was hindered significantly due to private functions in the JavaScript classes. These functions do not start out in the objective set, and are only added if they are called. Thus, the algorithms do not optimize to cover the branches of these functions, and if they never end up calling it, all of the branches from these functions will go uncovered.

Finally, the test cases generated by the algorithms were somewhat difficult to understand. A potential improvement would be to come up with methods to promote shorter and more readable test cases. The preference sorting method in DynaMOSA, also integrated in DynaMOSAPCSEA attempt to minimize the test case length as a secondary objective. This means, if two cases that achieve the same coverage are found, the shortest test case is kept in the archive. However, as we can see from the example test case generated by DynaMOSAPCSEA shown in Section 5, there is still room for improvement. This would not impact the performances of the algorithms, however, it would be valuable for developers using test case generation tools to create or augment test suites for their software.

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] 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. https://doi.org/10.1007/s10664-013-9249-9

[3] S. Brisset and F. Gillon. 2015. 4- Approaches for multi-objective optimization in the ecodesign of electric systems. In *Eco-Friendly Innovation in Electricity Transmission and Distribution Networks*, Jean-Luc Bessède (Ed.). Woodhead Publishing, Oxford, 83–97. https://doi.org/10.1016/B978-1-78242-010-1.00004-5

[4] Kalyanmoy Deb. 2011. *Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction.* Springer London, 4–5. https://doi.org/10.1007/978-0-85729-652-8_1

[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. https://doi.org/10.1109/4235.996017

[6] Kalyanmoy Deb and J. Sundar. 2006. Reference Point Based Multi-Objective Optimization Using Evolutionary Algorithms. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (Seattle, Washington, USA) *(GECCO '06)*. Association for Computing Machinery, New York, NY, USA, 635–642. https://doi.org/10.1145/1143997.1144112

[7] 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

[8] Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. 2016. Performance comparison of NSGA-II and NSGA-III on various many-objective test problems. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. 3045–3052. https://doi.org/10.1109/CEC.2016.7744174

[9] V. Khare, X. Yao, and K. Deb. 2003. Performance Scaling of Multi-objective Evolutionary Algorithms. In *Evolutionary Multi-Criterion Optimization*, Carlos M. Fonseca, Peter J. Fleming, Eckart Zitzler, Lothar Thiele, and Kalyanmoy Deb (Eds.). Springer Berlin Heidelberg, 376–390. https://doi.org/10.1007/3-540-36970-8_27

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

[11] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* 26 (2004), 369–395. https://doi.org/10.1007/s00158-003-0368-6

[12] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156. https://doi.org/10.1002/stvr.294 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.294

[13] 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

[14] 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

[15] 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. https://doi.org/10.1007/978-3-031-21251-2_5

[16] Kinza Yasar. August 2022. *What is Software Testing?* https://www.techtarget.com/whatis/definition/software-testing