Is PSO a valid option for search-based test case generation in the context of dynamically-typed languages?

Diego Viero

Supervisor(s): Annibale Panichella, Mitchell Olsthoorn, Dimitri Stallenberg

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: Diego Viero
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Dimitri Stallenberg, Sicco Verwer

ABSTRACT

In recent decades, automatic test generation has advanced significantly, providing developers with time-saving benefits and facilitating software debugging. While most research in this field focused on search-based test generation tools for statically-typed languages, only a few have been adapted for dynamically-typed languages. The larger search-space, generated by the dynamic allocation of types, causes standard search-based algorithms to not be as efficient in this domain and requiring a different approach. Existing algorithms like NSGA-II, MOSA and DynaMOSA have been employed to address this problem, but exploring different approaches may yield better results. That is why this paper proposes a different procedure based on an adaptation of the particle swarm optimization algorithm (PSO). The adaptation was evaluated using the SynTest framework, showing that DynaMOSA achieves better results than the presented approach, both when comparing the PSO adaptation with and without DynaMOSA features, and when comparing the base DynaMOSA algorithm with PSO adapted to include DynaMOSA ingredients.

1 INTRODUCTION

Software testing consists in verifying that the code written behaves as expected and can be trusted so that more functionalities can be built upon it. Writing meaningful and useful tests normally consumes precious time, which developers often prefer spending implementing new features or optimizing existing ones [11]. Furthermore, with the increasing complexity of modern day applications, designing tests that have high-coverage is becoming a complicated task. Because of these reasons, in the past years there has been a significant growth in the field of automated test cases generation [3]. Such field focuses on creating applications that analyze the code written and automatically generate meaningful test cases, substantially improving the developer's workflow and reducing the time spent debugging.

Currently, the main approaches used to generate tests are random testing [4], symbolic execution [2], and search-based testing [14]. The latter has been shown to achieve better results for most situations [1, 8, 18] and the approach presented in this paper falls under that category. The concept behind search-based testing is to first generate test cases randomly, and then use meta-heuristics algorithms to evolve them over generations. The evolutionary step continues until the set of tests, which will eventually represent the final test-suite, achieves high coverage of the program.

Despite significant advancements in this field over the past few decades, most research has focused on statically-typed languages, neglecting their dynamically-typed counterparts [12]. One of the main challenges introduced by dynamically-typed languages is the larger search space caused by the variables' types flexibility. This and other complications keep tools designed for statically-typed languages from creating adequate solutions for both kinds [20].

Examples of algorithms that were adopted for this domain and that are used as comparison for the implementation discussed include: NSGA-II [7] and DynaMOSA [16].
NSGA-II uses the dominance of solutions and crowding distance as heuristics to select the parents used to generate the offspring for the next generation. To adapt NSGA-II to the test-case generation domain, three mutation operators and a custom crossover operator were introduced on top of the basic algorithm [15].
Dynamic many-objective sorting algorithm [16] is an extension of NSGA-II that was developed specifically for the automated test-case generation domain. The algorithm introduces two main heuristics, the first one being the dependency between *conditional branches*, which are used to select only the solutions that cover conditionally independent branches for the current generation. And the second one being the *preference criterion* function, used to assign ranks to the solutions based on a combination of branch distance and approach level, the ranks are then used as a mean to select the population for the next generation.

As previously mentioned, search-based test generation focuses on using algorithms, mainly evolutionary and meta-heuristics, to evolve a set of randomly generated tests. Various meta-heuristics approaches have been tested and proven to be a good method to generate test-cases [22]. The main characteristic that makes them so useful is the exploration of the search space, which is fundamental in the test-case generation domain, especially for dynamically-typed languages. Recent studies showed that DynaMOSA outperforms other meta-heuristics also in different tools (EvoSuite, Pynguin) and programming language (Java and Python) [13, 17].

This paper focuses specifically on an adaptation of the *Particle Swarm Optimization algorithm* (PSO) [10] for search-based test case generation. PSO is an optimization algorithm inspired by the social behaviour of bird flocks. Initially it generates a population of particles representing candidate solutions, then the position of each particle is iteratively updated based on the best-known position of the particle itself and the best global position of all particles. The social behaviour of this algorithm allows PSO to efficiently explore the search space and find optimal solutions for complex optimization problems.

Some of the methods present in the base algorithm were altered in order to adapt them to the test-case generation domain; this includes the velocity and position update, which had to be adapted to fit the new particles' encoding.

To test the adapted algorithm, we have implemented and adapted PSO within the SynTest framework[1]. SynTest is a framework created to automatically generate JavaScript unit-level test cases, it was chosen because it already contains implementations for both NSGA-II and DynaMOSA and represents a good starting point to write our adaptation.

To assess the performance of PSO, we have carried out an empirical study on a benchmark containing popular JavaScript projects, that were selected to create a good representation of the language's syntax and code styles.

To understand the quality of the results, the presented adaptation is compared to DynaMOSA and to a basic version of PSO without DynaMOSA features.

The obtained results show that DynaMOSA performs slightly better than the presented PSO adaptation, while the comparison

[1] https://www.syntest.org/

between PSO with and without DynaMOSA ingredients displays relatively higher coverage for the algorithm with DynaMOSA features. The outcomes of the experiment confirm the superiority of the approach used by the DynaMOSA algorithm.

In order to independently replicate and validate the findings, experiments, and results presented in the paper, a replication package is made available in this GitHub repository.

2 BACKGROUND

The purpose of this chapter is contextualise this study and introduce background concepts, namely single and multi-objective optimization; numeric evolutionary optimization, including how crossover and mutation work; how the NSGA-II optimization algorithm evolves solutions over time; the adaptation of multi-objective optimization to the test-case generation domain, including how DynaMOSA works and how PSO explores the search space to find better solutions in the numeric domain.

2.1 Numeric multi-objective optimization

In the mathematical field, optimization refers to the process of maximizing or minimizing a function, known as the *objective function*, by iteratively selecting better input values for it. Optimization problems can be divided in two different categories, *single-objective problems* and *multi-objective problems*. The first category consists of problems that have only one objective function, while the second includes problems with two or more objective functions.

When dealing with multi-objective problems, comparing two solutions may not be as straightforward: given two inputs x_1, x_2 and two objective functions f_1, f_2 that should be maximized, it might happen that $f_1(x_1) > f_1(x_2)$ and $f_2(x_1) < f_2(x_2)$, thus we cannot conclude whether x_1 or x_2 is a better solution. In multi-objective optimization, the quality of the solutions is measured using the *dominance*. Dominance, denoted as $X \prec Y$, indicates that solution X is superior to at least one objective compared to Y, while being at least as good for the remaining objectives. In other words, a solution is considered dominant if it offers improved performance in at least one objective without compromising performance in any other objective. Dominance serves as a crucial concept for identifying non-dominated solutions in multi-objective optimization problems.

2.2 Numeric evolutionary optimization

Evolutionary algorithms are computational optimization techniques inspired by the process of natural selection. By emulating the principles of evolution, these algorithms generate and evolve a population of candidate solutions iteratively, aiming to find optimal or near-optimal solutions for complex problems. Through genetic operators like mutation and crossover, the population continually improves over successive generations. This iterative process allows evolutionary algorithms to effectively explore solution spaces and find solutions that may be difficult to discover through traditional methods.

In most evolutionary algorithms, the initial step involves *encoding the candidate solutions*, converting them into vectors known as chromosomes. These chromosomes are then utilized to produce an offspring for the next generation, using crossover and mutation.

2.2.1 Crossover. Given two chromosomes C_1 and C_2 with length l, the crossover operation selects a random index $i \in [0, l]$, then swaps the parts of C_1 and C_2 after that index to create two new children. *e.g.,* $C_1 = [0.4, 0.2, 1.0, 0.7]$, $C_2 = [0.2, 0.5, 0.6, 0.3]$ and $i = 1$ the crossover operation generates the two new chromosomes $C_3 = [0.4, 0.2, 0.6, 0.3]$ and $C_4 = [0.2, 0.5, 1.0, 0.7]$.

2.2.2 Mutation. The mutation operator is used to introduce diversity in the generated population, it operates by randomly changing a value in a chromosome with a certain probability. Given a chromosome $C = [0.4, 0.2, 1.0, 0.7]$, the mutation operator has a probability of $1/l$, with l being the length of the chromosome, of mutating each value in the encoding. Thus if that condition is met at the second position of C, the resulting chromosome will be $C' = [0.4, \mathbf{0.9}, 1.0, 0.7]$.

2.3 NSGA-II

Non-dominated Sorting Genetic Algorithm II is a multi-objective genetic algorithm developed by Deb *et al.* [7]. The algorithm initially generates a set of random solutions of size N, then it iteratively evolves them to find better test cases. To obtain fitter solutions, it selects some of the existing tests as parents using binary tournament selection, then it uses them to generate an offspring by applying crossover and mutation. Once an offspring of size N is generated, two heuristics are used to select the fittest candidates from the union of the current population and the offspring. (i) The dominance of other solutions to the current one, where the tests that are the least dominated have a higher chance of being chosen. (ii) Crowding distance, a parameter that reduces the probability of selecting two nearby solutions as parents. Meaning that the further a non-dominated solution is to the others, the higher the chance it has of being selected.

The N fittest candidates are used as the population for the next generation.

2.4 Multi-objective optimization for test-case generation

Writing tests is commonly considered a time-consuming and effort-intensive step in development, that's why, in the past decades, researchers have explored various techniques to automate this process. Some approaches include random testing [4], symbolic execution [2], and meta-heuristics, also known as search-based testing [14]. Among these options, search-based testing was proven to be more effective, achieving greater coverage and bug detection [1, 8, 18].

Its procedure relies on randomly generating a set of test cases, then using meta-heuristics algorithms to evolve it to a test-suite that (possibly) achieves maximum coverage for our program. This approach converts the test-generation issue to a multi-objective optimization problem, where each conditional branch in the program tested represents an objective, and the population of tests is optimized towards covering more objectives. Each solution is evaluated based on its coverage, where tests with a higher coverage are more likely to be chosen to generate the offspring for the next generation.

2.4.1 DynaMOSA. Dynamic many-objective sorting algorithm [16] is an extension of NSGA-II that introduces the concepts of

preference criterion and *dynamic selection* of the target branches to cover.

Dynamic selection is based on the structural dependencies among the branches, a *conditional branch* b_2 is conditionally dependent to a branch b_1 when b_2 can only be satisfied if b_1 is satisfied. After generating the initial solutions, DynaMOSA selects only the branches that do not have conditional dependencies. These solutions are inserted into a set U^* and ranked using the *preference criterion* function.

Such method assigns to each test an objective score based on the *branch distance* and *approach level* of each uncovered target. The approach level represents the number of control dependencies separating the execution trace from the target, while the branch distance represents the variable values evaluated at the conditional expression where the execution diverges from the target[16]. Solutions with the lowest objective score for a given target are assigned rank 0. Solutions not selected by this method are ranked using NSGA-II starting from rank 1.

Once all ranks are assigned, DynaMOSA creates a fixed-size set as population for the next generation. The set is initially filled with solutions of rank 0, if there are not enough solutions to fill the set, the algorithm uses the solutions with rank 1, and so on until the set is full.

Additionally, the algorithm maintains an archive to store the best tests for each covered branch based on preference criterion. Once a solution is found for an objective, that specific branch is excluded from the search. This process saves computation time as the algorithm only needs to compute a subset of all solutions. At each iteration, U^* is updated including solutions dependent on the conditional branches covered by the newly generated tests. This dynamic selection of solutions is what gives DynaMOSA its name and allows it to obtain faster results compared to its alternatives.

2.5 Particle Swarm Optimization

Particle Swarm Optimization [10] is an evolutionary algorithm inspired by the social behaviour of bird flocks. Its approach starts by generating random particles over the search space, where each particle represents a candidate solution. Then, it computes a velocity vector for each particle and uses it to iteratively update their positions, moving them closer to our objective.

The new position is calculated as

$$X_{id}(t+1) = X_{id}(t) + V_{id}(t+1)$$

where $X_{id}(t)$ represents the position of particle id at time t, and V represents the velocity vector. The formula used to calculate the velocity is the following:

$$V_{id}(t+1) = \omega V_{id}(t) + c_1 r_1 (P_{id}(t) - X_{id}(t)) + c_2 r_2 (P_{gd}(t) - X_{id}(t))$$

Here ω is the inertia weight constant and it is used to control the speed and direction of the search. c_1 is the cognitive coefficient and determines the exploration of the current particle. c_2 is the social coefficient and regulates how much the particle should follow the flock. r_1 and r_2 are random values; $P_{id}(t)$ is the best solution found by particle id at iteration t and $P_{gd}(t)$ is the best solution found by all particles at iteration t.

The original implementation of the algorithm that we just described presents some limitations. Firstly, PSO was developed to optimize a static set of objectives, which in the context of search-based testing represents a drawback, since there exist structural dependencies among targets that should be considered when deciding which objectives to optimise [16]. Secondly, the algorithm is meant to address numerical problems, thus we cannot apply it directly to the context of test-generation without first adapting it. The work presented in the next chapter explains how PSO was adapted to be utilized for search-based test generation.

3 APPROACH

This section discusses the approach used to adapt PSO for its usage in the test-generation domain. As explained in the previous section, PSO was originally thought for numeric optimization problems, and requires some adaptations to be used in this context. This chapter first introduces the adaptations from single-objective to multi-objective optimization using the method proposed by Coello Coello *et al.* [5], then discusses the intuition behind the solutions encoding and how they affect the velocity and position update of the algorithm. Afterwards, it covers the problems that emerge from the proposed adaptation, and introduces the changes needed for its usage in the SynTest framework [1], finally it presents an alternative implementation that was adapted to include DynaMOSA features. The pseudo-code for the adaptation with DynaMOSA ingredients can be found in algorithm 1, while the code for the position update can be found in algorithm 2.

3.1 Adapting the algorithm to multi-objective domain

The majority of the PSO approach remains unaffected by the addition of objective functions, with only the selection of the local (P_{id}) and global (P_{gd}) best solutions requiring adjustments. This adaptation draws from the approach described in *"MOPSO: A Proposal for Multiple Objective Particle Swarm Optimization"* by Coello Coello *et al.* [5].

To update P_{id}, the current particle X_{id} is compared to P_{id}. If $X_{id} \prec P_{id}$, P_{id} is updated with the value of X_{id}; otherwise, it remains unchanged.

Selecting P_{gd} involves creating an archive A that stores all non-dominated solutions, where P_{gd} is chosen from A using a weighted probability approach. Each solution in A has a probability of being selected as P_{gd} inversely proportional to the number of particles it dominates in the population.

3.2 Solutions encoding

Before explaining how the algorithm was adapted for this specific domain, we need an appropriate encoding of the solutions. Given an arbitrary program for which we are trying to generate tests, each class in the program can be represented as a tree $C = \langle V, E \rangle$. Each vertex $m_i \in V$ symbolizes either a method or a variable computed using a method of the class, where the root $r \in V$ is always the class' constructor. The vertices are connected through directed edges, where vertices m_1 and m_2 have an edge $e(m_1, m_2) \in E$, going from m_1 to m_2, if m_2 is used inside m_1, thus representing a conditional dependency between the methods. Given this representation of the

program's classes, we can now adapt it to represent the method calls performed in each generated test.
A test T is defined by a set of trees $\overline{I}$, where each $I \in \overline{I}$ represents an instantiation of a class in the program tested. A tree I has the same characteristics as the previously described tree $C = \langle V, E \rangle$, but I only contains the vertices of the methods called inside of T.

3.3 Velocity computation

The encoding adaptation explained in the previous section introduces complications for the PSO formula presented in 2.5, specifically for X_{id}, P_{id} and P_{gd}. Because particles now consist of classes encoding instead of real numbers, we cannot use them directly to compute the particle's velocity for the next generation. To address this problem, this implementation replaces the real values in the formula with the *objective score* of the current test, introduced in 2.4.1, representing how far a test is from covering an objective.
"*The distance is based on the number of control dependencies that separate the execution trace from the target (approach level) and on the variable values evaluated at the conditional expression where the execution diverges from the target (branch distance).*" [16]
This adaptation now yields a velocity vector $V_{id}(t)$ consisting of real values, where $v_i \in V_{id}(t)$ equals 0 if the particle covers the conditional branch at dimension i.
Even though the velocity computation was slightly changed, the main concept of velocity from PSO still applies, high velocity values represent particles further from the desired objective, and low velocity values represent particles close to the objective.

3.4 Position update

The encoding adaptation affects more than just the velocity update. It also requires altering the position update formula seen in 2.5. In the proposed approach, we maintain the core concept of the standard PSO algorithm: updating the positions of particles that are farther from the objective more than those that are closer to it. By analyzing a particle's velocity we can determine how far it is from covering each objective. We can then use this parameter in order to apply mutation to the generated test at different levels of its trees $\overline{I}$, thus controlling how much it is altered in the next iteration. If the velocity is high, the mutation step is performed at a higher node of the tree, therefore changing all of the node's descendants as well. Similarly, if the velocity is low, the tree is mutated at a lower level, thus keeping most of the test's structure. The mutation of test cases either adds, deletes or changes a statement. The method uses uniform mutation with probability of $1/n$, where n is the number of statements in the test case to mutate [9].

3.5 Problems with described adaptation

Although the implementation explained in the previous section is promising, SynTest[1], the tool used to test this PSO adaptation, lacks the necessary functionality to control the level at which each particle can be mutated, instead it applies mutation at a random level of the encoding.
To account for this problem, the position update adaptation explained in the previous section is slightly altered as shown in algorithm 2. Instead of selecting the level of mutation, the method is modified to randomly mutating the particle based on the velocity. Given a particle X_{id} and the corresponding velocity vector V_{id}, with values $\{v_0, v_1, ..., v_n\} \in V_{id}$, the method uses the approach shown in algorithm 2: first, the vector is normalized to values in range $[0, 1]$ using the highest (max_v_i) and lowest (min_v_i) values of all velocities (line 3); then, each value v'_i of the normalized vector V'_{id} is compared to a randomly generated number $r \in [0, 1]$ (line 6 and 7). The particle is then mutated if $r > v_i$ (line 8).

3.6 DynaMOSA features

The previous section introduced an adaptation that presents a potential threat. While mutating a particle based on its velocity is a reasonable approach, the method lacks control over the resulting mutation. As the presented adaptation relies on probability, there is a chance that a particle might generate a test with lower coverage in the next generation.
To address this concern, this section presents a further improvement referred to as DynaMOSAPSO, which incorporates features from DynaMOSA. Algorithm 1 outlines this adaptation, which can be divided in three steps: (i) generating the mutated population $\overline{P}$ (lines 5 to 9) following the same procedure explained earlier in this chapter; (ii) merging the mutated population $\overline{P}$ with the original one P (line 10) to create a set containing the original and mutated population; (iii) feeding the new set to DynaMOSA (line 10).
The intuition behind this adaptation is to take advantage of the search-space exploration quality of PSO, while maintaining the test coverage. In the worst-case, where all particles are mutated and their coverage is decreased for all conditional branches, DynaMOSA will only select solutions from the original population, preserving the previous generation's coverage.
To differentiate the two implementations, from this point on we will refer to the implementation without DynaMOSA features as SearchBasedPSO and to the implementation with them as DynaMOSAPSO. The pseudo code for SearchBasedPSO is not provided, but the implementation is the same as shown in algorithm 1, with the only difference being that the mutated population $\overline{P}$ is set directly as the next generation's population, without applying the DynaMOSA routine merging it with the original population, as seen in line 10.

4 STUDY DESIGN

To evaluate the quality of the proposed implementation and be able to compare it to other alternatives, we perform an empirical analysis using the SynTest[1] framework as our main testing tool. This section discusses the different steps taken to perform such analysis, including the description of the tool used and the system on which the experiment was run.
First we introduce the research questions; then we discuss the tools used to obtain the results; afterwards, we discuss the different configurations we considered; and finally we explain how the experiment was performed.

4.1 Research questions

As stated in section 3.6, the implementation without DynaMOSA features is referred to as SearchBasedPSO, while the one that includes them is called DynaMOSAPSO. To determine the quality

Algorithm 1: PSO adaptation with DynaMOSA features

input :
$U = \{u_1, \ldots, u_m\}$ Set of coverage targets
Population size N
output : Generated test-suite

```
S ← RANDOM-POPULATION(N);
while not(search_budget_consumed) do
    S̄ ← ∅;
    A ← GET-NON-DOMINATED-FRONT(S);
    for X_id ← S do
        P_id ← SELECT-PBEST(X_id);
        G_id ← SELECT-GBEST(X_id, A);
        V_id ← UPDATE-VELOCITY(X_id, V_id, P_id, G_id);
        S̄ ← S̄ ∪ UPDATE-POSITION(X_id, V_id);
    S ← MOSA(S̄ ∪ S);
return S;
```

Algorithm 2: Update positions adaptation

input :
Current particle: X_{id}
Corresponding velocity V_{id}
output : Possibly mutated particle: $\overline{X_{id}}$

```
min_v_i ← MIN({V_0, ..., V_N});
max_v_i ← MAX({V_0, ..., V_N});
V'_id ← MIN-MAX-NORMALIZATION(min_v_i, max_v_i, V_id);
X̄_id ← X_id;
for v'_i ← V'_id do
    r ← RANDOM-NUMBER-IN-RANGE(0,1);
    if r > v'_i then
        X̄_id ← Mutate(X̄_id);
return X̄_id
```

of the presented approach we perform an empirical analysis in order to answer the following questions:

4.1.1 RQ1. How does DynaMOSAPSO perform compared to the default SearchBasedPSO implementation?

4.1.2 RQ2. How does the DynaMOSAPSO perform compared to the original DynaMOSA algorithm?

4.2 Benchmark

SynTest[1] is a state of the art framework created to automatically generate JavaScript unit-level test cases, it is used in this paper as the primary tool to test our implementation. The framework was chosen because it already contains implementations for NSGA-II and DynaMOSA, and represents a good starting point to write our adaptation.
In order to determine the quality of the presented algorithm, we make use of a benchmark built by the SynTest[1] team and designed specifically for this purpose. The benchmark consists of five popular JavaScript projects: Express [2], Commander.js [3], Moment.js [4], JavaScript Algorithms [5] and Lodash [6].

This set of projects depicts a good representation of the syntax and the different code styles present in the JavaScript language, making it a great tool to evaluate our adaptation.
A more in depth description of the benchmark and explanation of why it is an appropriate choice to test the implementation can be found in chapter 4 of *"Guess What: Test Case Generation for JavaScript with Unsupervised Probabilistic Type Inference"* [20].

All of the files from Moment.js and one file from Express were removed from the version of the benchmark used in this paper as they were giving errors for the presented algorithms. Even though less files are used for comparison, the benchmark is still considered a valid representation of the JavaScript language.

4.3 Configurations

To test SearchBasedPSO, the algorithm was executed using a simple objective manager that tries to optimize all possible objectives, this algorithm will be referred as PSO in results table 1. While to test DynaMOSAPSO, a structural-uncovered objective manager was used, which takes into account only conditionally independent objectives at each generation.
In order to answer RQ1 and RQ2, we tested the following two configurations using SynTest framework:

(i) DynaMOSAPSO vs PSO
(ii) DynaMOSAPSO vs DynaMOSA

4.3.1 Parameters. Here we report the parameters used for the sake of replicability: the population size was set to 50 for all configurations; even though the article published by Piotrowski *et al.* [19] mentions that a population of 50 may not be optimal for PSO, running preliminary studies with population sizes of 50, 100, 150, 200 and 250 did not rise significant differences in the results. Thus, to keep the comparison as fair as possible, we decided to use the same population size for all configurations. The default mutation rate of the system is $1/n$ where n= "statements in test case", although for PSO the rate was adapted as explained in section 3.5 to i/n, where i= a random number proportional to the velocity of each particle, with $i \in [0, |V_{id}|]$. The search time was set to 60 seconds, while the crossover probability used was 0.7 for all configurations.

4.3.2 Hyperparameters. Hyperparameters are often fine-tuned using optimization methods, in order to find the best combination for the current domain. Unfortunately, the presented implementations' hyperparameters can't be optimized because of time constraints. The full benchmark takes ~45 minutes to run for one run, running the full experiment for one configuration takes (45 min × 10 runs) / (60 min) = ~7.5 hours of consecutive runs to finish and we would need to run it for each possible combination of hyperparameters and for the three PSO configurations. That is why we decided to use the hyperparameters that have been shown to work best for

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

the numeric approach of the algorithm, which in the case of PSO are: $\omega = 0.5$, $c_1 = 0.25$ and $c_2 = 0.25$.

4.4 Experiment protocol

Because the metrics used by the benchmark are time-based, using different machines to evaluate different algorithms would produce incomparable results. Thus here we provide a description of the system used for the evaluation process of the different algorithms. The experiment was performed on a system with 2 AMD EPYC 7H12 processors (64 cores, 128 threads, 3.3GHz each) with 512GB of RAM where 100 cores ran in parallel.

Because of the stochastic nature of SynTest, no two runs of the benchmark can be directly compared, as one algorithm could have had a more favorable initial population than the other, thus leading us to false conclusions. One option to solve this issue could be to use the same seed for the random generation, so that the algorithms start with the same set of tests. Although, this approach presents a complication too, since the set could be more favorable for the evolution of one approach compared to the other.
In order to perform a legitimate comparison, the benchmark was run ten times for each configuration, storing the final coverage achieved and coverage over time for each file, and computing the average of the ten runs for the two parameters. Once all results were obtained, we applied the unpaired Wilcoxon signed-rank test [6] with a 0.05 threshold, which gives us an understanding of how different two data distributions are. In addition, we apply the Vargha-Delaney $\hat{A}_{12}$ statistic [21] to determine the difference in magnitude between the two data distributions.

The results produced are presented in the following section in the form of tables computed using the R programming language.

5 RESULTS

In this section, we analyze and discuss the results of the empirical study presented in table 1. The comparison metric used is the percentage of branch coverage for the tested files. The first two columns show the project and file names, while the remaining columns display the branch coverage obtained for each algorithm and the corresponding statistical data. The highest coverage achieved for each file is highlighted in the aforementioned table.
Table 2 provides an overview of the results obtained from comparing the average branch coverage. The *#Win* column indicates the number of times the left configuration had a higher coverage than the right one, *#Lose* denotes the times it had a lower coverage, and *#No Diff.* represents the instances where the coverage was the same. The columns are further divided into categories (*Negligible, Small, Medium* and *Large*) based on the $\hat{A}_{12}$ effect size.
The following files have been excluded from the table because their coverage was 0% across all tested algorithms, hence not providing any additional information to the results: `articulationPoints.js`; `bellmanFord.js`; `bfTravellingSalesman.js`; `detectDirectedCycle.js`; `detectUndirectedCycle.js`; `eulerianPath.js`; `floydWarshall.js`; `hamiltonianCycle.js` and `stronglyConnectedComponents.js`. The reason behind the lack of coverage is that these files require input parameters in the form of graphs, which are often challenging to generate.

5.1 Results for RQ1

Table 2 provides an overview of the comparison between DynaMOSAPSO vs SearchBasedPSO in terms of average branch coverage. We can observe that the algorithm with DynaMOSA features performed relatively better than the basic adaptation. Out of 27 reported files, DynaMOSAPSO achieved a higher coverage than SearchBasedPSO in 6 files, a lower coverage in 1 file, and the same coverage in the remaining 20 files. The average coverage for DynaMOSAPSO across all reported files is of 54.87%, while for SearchBasedPSO it is 53.89%. The file with the largest coverage difference between the two algorithms is `transform.js`, where DynaMOSAPSO achieved a higher average coverage of 12.50% than SearchBasedPSO.
From the results obtained we can conclude that applying DynaMOSA features to the basic adaptation improves the achieved coverage of the algorithm.

5.2 Results for RQ2

When comparing DynaMOSAPSO and DynaMOSA, we observe even more similarities in the results, with DynaMOSAPSO performing either as good or worse in all tested files. The two algorithms achieved the same average coverage for 23 out of the 27 tested file, DynaMOSA improved the coverage for four files, with `help.js` and `response.js` resulting in a large difference in magnitude. The average coverage of the reported files is of 54.87% for DynaMOSAPSO and 55.24% for DynaMOSA. The file with the largest coverage difference between the two algorithms is `breadthFirstSearch.js`, where DynaMOSA achieved a higher average coverage of 6.25%.
One possible interpretation of these results is that the initial phase of the DynaMOSAPSO algorithm, which involves particle mutation, may not generate better solutions in subsequent generations. This step could potentially consume valuable search time, limiting the number of iterations and preventing the algorithm from finding better solutions.

6 THREATS TO VALIDITY

This section discusses the potential threats to the validity of this paper. The benchmark used to evaluate the proposed adaptation contains different open-source projects, written in JavaScript, having varying sizes and applications. Such factor allows the results obtained to be a valid representation of the proposed adaptation's quality.
Because DynaMOSA, NSGA-II and PSO initially generate a population using random seeds, the evaluation was executed 10 times for each algorithm. Once we obtained the final results, the average was computed and used to draw conclusions. This step prevented false comparisons between the results where one seed might have generated a better starting point for one configuration compared to another.

7 RESPONSIBLE RESEARCH

The topics discussed in this paper do not raise any significant ethical concern, the tools [7] [8] [9] and adaptation presented [10] are open-source

[7] https://github.com/syntest-framework/syntest-core.git
[8] https://github.com/syntest-framework/syntest-javascript.git
[9] https://github.com/syntest-framework/syntest-javascript-benchmark.git
[10] https://github.com/Diego-Viero/syntest-core.git

Benchmark	File name	DynaMOSA	DynaMOSAPSO	PSO	DynaMOSAPSO vs DynaMOSA		DnyaMOSAPSO vs PSO	
					p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
Commander.js	help.js	50.00%	48.48%	48.48%	0.005	0.2 (Large)	0.447	0.6 (Small)
	option.js	50.00%	50.00%	38.89%	0.824	0.47 (Negligible)	0.022	0.78 (Large)
	suggestSimilar.js	71.88%	71.88%	71.88%	0.582	0.44 (Negligible)	0.754	0.54 (Negligible)
Express	query.js	66.67%	66.67%	66.67%	NA	0.5 (Negligible)	0.077	0.65 (Small)
	request.js	32.61%	32.61%	30.43%	1.000	0.5 (Negligible)	0.025	0.75 (Large)
	response.js	19.57%	18.48%	17.39%	0.026	0.205 (Large)	0.054	0.755 (Large)
	utils.js	42.39%	41.30%	41.30%	0.693	0.45 (Negligible)	0.099	0.68 (Medium)
	view.js	37.50%	37.50%	37.50%	1.000	0.5 (Negligible)	0.368	0.55 (Negligible)
JS Algorithms	breadthFirstSearch.js	18.75%	12.50%	12.50%	0.398	0.4 (Small)	0.681	0.45 (Negligible)
	depthFirstSearch.js	0.00%	0.00%	8.33%	0.681	0.45 (Negligible)	0.398	0.4 (Small)
	kruskal.js	20.00%	20.00%	20.00%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	prim.js	16.67%	16.67%	16.67%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	Knapsack.js	57.50%	57.50%	57.50%	NA	0.5 (Negligible)	0.368	0.55 (Negligible)
	KnapsackItem.js	50.00%	50.00%	50.00%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	Matrix.js	7.89%	7.89%	7.89%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	CountingSort.js	57.14%	57.14%	57.14%	1.000	0.5 (Negligible)	0.368	0.45 (Negligible)
	RedBlackTree.js	29.41%	29.41%	29.41%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
Lodash	equalArrays.js	83.33%	83.33%	79.17%	1.000	0.495 (Negligible)	0.037	0.74 (Large)
	hasPath.js	100.00%	100.00%	100.00%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	random.js	100.00%	100.00%	96.43%	0.167	0.4 (Small)	0.161	0.66 (Small)
	result.js	80.00%	80.00%	80.00%	0.583	0.45 (Negligible)	0.368	0.55 (Negligible)
	slice.js	100.00%	100.00%	100.00%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	split.js	87.50%	87.50%	87.50%	NA	0.5 (Negligible)	NA	0.5 (Negligible)
	toNumber.js	65.00%	65.00%	65.00%	NA	0.5 (Negligible)	0.368	0.55 (Negligible)
	transform.js	91.67%	91.67%	79.17%	1.000	0.5 (Negligible)	0.045	0.76 (Large)
	truncate.js	55.88%	55.88%	55.88%	0.167	0.4 (Small)	NA	0.5 (Negligible)
	unzip.js	100.00%	100.00%	100.00%	0.368	0.55 (Negligible)	0.167	0.6 (Small)

Table 1: Average branch coverage of different algorithms tested using SynTest benchmark

Comparisons	# Win				# No Diff			# Lose			
	Negl	Small	Medium	Large	Negl	Small	Medium	Negl	Small	Medium	Large
DynaMOSAPSO vs PSO	-	1	-	5	16	3	1	-	1	-	-
DynaMOSAPSO vs DynaMOSA	-	-	-	-	21	2	-	1	1	-	2

Table 2: Statistical results w.r.t. branch coverage

and accessible by everyone. A possible implication of this paper would arise if SynTest developers choose to incorporate the presented adaptation as an optional module in the framework. In such scenario, end users could select our algorithm to test their code, relying on our adaptation to determine the quality of their software and taking actions based on the produced results.

In terms of experiment reproducibility, all methods and tools used are open-source and available online, although it is important to note that while the reported results should provide an accurate representation of the algorithm's quality, the process of test generation is stochastic and may not always yield identical results.

8 CONCLUSION AND FUTURE WORK

In this paper we presented two adaptations of the PSO algorithm, one with DynaMOSA features (DynaMOSAPSO) and one without (SearchBasedPSO), for search-based test case generation in the context of dynamically-typed languages. DynaMOSAPSO was compared both to SearchBasedPSO and to DynaMOSA, showing that the DynaMOSA features applied to DynaMOSAPSO allow the algorithm to obtain better results than the adaptation without them. Furthermore DynaMOSA showed to still be a better approach than the proposed DynaMOSAPSO algorithm, proving to be the best approach for test-case generation in the context of dynamically-typed languages.

Due to time constraints, the proposed adaptation utilized the default hyperparameters of PSO for numeric problems. However, we believe that conducting an optimization on these values could potentially yield improved results. As mentioned in the study design section, the experiment employed a population size of 50 particles; While preliminary studies assessed various values, these experiments focused solely on the lodash [6] repository of the benchmark. Additionally, as outlined in 3.5, the current state of SynTest framework [1] lacks control over the tree level at which mutation should be applied. Therefore, the exploration of different population sizes for this domain, the optimization of hyperparameters and the implementation of a mutation function that can be applied at different levels of the encoding's tree is left as future work.

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)*. 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657

[3] George Candea and Patrice Godefroid. 2019. *Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances*. Springer International Publishing, Cham, 505–531. https://doi.org/10.1007/978-3-319-91908-9_24

[4] T. Y. Chen, H. Leung, and I. K. Mak. 2005. Adaptive Random Testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, Michael J. Maher (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 320–329.

[5] C.A. Coello Coello and M.S. Lechuga. 2002. MOPSO: a proposal for multiple objective particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, Vol. 2. 1051–1056 vol.2. https://doi.org/10.1109/CEC.2002.1004388

[6] W.J. Conover. 1999. *Practical Nonparametric Statistics*. Wiley. https://books.google.nl/books?id=n_39DwAAQBAJ

[7] 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

[8] Gordon Fraser and Andrea Arcuri. 2013. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering* 20, 3 (Nov. 2013), 611–639. https://doi.org/10.1007/s10664-013-9288-2

[9] 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

[10] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[11] Divya Kumar and K.K. Mishra. 2016. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science* 79 (2016), 8–15. https://doi.org/10.1016/j.procs.2016.03.003 Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[12] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *Search-Based Software Engineering*, Aldeida Aleti and Annibale Panichella (Eds.). Springer International Publishing, Cham, 9–24.

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

[14] 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

[15] Mitchell Olsthoorn and Annibale Panichella. 2021. Multi-objective Test Case Selection Through Linkage Learning-based Crossover. arXiv:2107.08454 [cs.SE]

[16] 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

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

[18] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. 2022. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering* 27, 7 (Sept. 2022). https://doi.org/10.1007/s10664-022-10207-5

[19] Adam P. Piotrowski, Jaroslaw J. Napiorkowski, and Agnieszka E. Piotrowska. 2020. Population size in Particle Swarm Optimization. *Swarm and Evolutionary Computation* 58 (2020), 100718. https://doi.org/10.1016/j.swevo.2020.100718

[20] 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* (Singapore, Singapore). Springer-Verlag, Berlin, Heidelberg, 67–82. https://doi.org/10.1007/978-3-031-21251-2_5

[21] 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. https://doi.org/10.3102/10769986025002101 arXiv:https://doi.org/10.3102/10769986025002101

[22] Lu Xiong and Kangshun Li. 2018. Research on Automatic Generation of Test Cases Based on Genetic Algorithm. In *Computational Intelligence and Intelligent Systems*, Kangshun Li, Wei Li, Zhangxing Chen, and Yong Liu (Eds.). Springer Singapore, Singapore, 351–360.