Higher Fault Detection Through Novel Density Estimators in Unit Test Generation

Panichella, Annibale; Olsthoorn, Mitchell

DOI
10.1007/978-3-031-64573-0_2

Publication date
2024

Published in
Search-Based Software Engineering - 16th International Symposium, SSBSE 2024, Proceedings

Citation (APA)
Panichella, A., & Olsthoorn, M. (2024). Higher Fault Detection Through Novel Density Estimators in Unit Test Generation. In G. Jahangirova, & F. Khomh (Eds.), *Search-Based Software Engineering - 16th International Symposium, SSBSE 2024, Proceedings* (pp. 18-32). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 14767 LNCS). https://doi.org/10.1007/978-3-031-64573-0_2

Higher Fault Detection Through Novel Density Estimators in Unit Test Generation

Annibale Panichella(✉) and Mitchell Olsthoorn

Delft University of Technology, Delft, The Netherlands
{a.panichella,m.j.g.olsthoorn}@tudelft.nl

Abstract. Many-objective evolutionary algorithms (MOEAs) have been applied in the software testing literature to automate the generation of test cases. While previous studies confirmed the superiority of MOEAs over other algorithms, one of the open challenges is maintaining a strong selective pressure considering the large number of objectives to optimize (coverage targets). This paper investigates four density estimators as a substitute for the traditional crowding distance. In particular, we consider two estimators previously proposed in the evolutionary computation community, namely the *subvector-dominance assignment* (SD) and the *epsilon-dominance assignment* (ED). We further propose two novel density estimators specific to test case generation, namely the *token-based density estimator* (TDE) and the *path-based density estimator* (PDE). Based on the CodeBERT model tokenizer, TDE uses natural language processing to measure the semantic distance between test cases. PDE, on the other hand, considers the distance between the source-code paths executed by the test cases. We evaluate these density estimators within EvoSuite on 100 non-trivial Java classes from the SF110 benchmark. Our results show that the proposed *path-based density estimator* (PDE) outperforms all other density estimators in enhancing mutation scores. It increases mutation scores by 4.26 % on average (with a max of over 60%) to the traditional crowding distance.

Keywords: software testing · search-based software engineering · test case generation · density estimators

1 Introduction

Many-objective evolutionary algorithms (MOEAs) have been used extensively in literature for automatically generating test cases [2,11,20,23]. These algorithms optimize multiple objectives (testing criteria) simultaneously, such as code coverage criteria (*e.g.,* lines, branches) and quality metrics (*e.g.,* mutation score). Previous studies have shown that MOEAs outperform single-objective algorithms in terms of both code coverage and fault detection [6,20,23]. MOEAs have led to various advancements in automated test case generation, such as (1)

G. Jahangirova and F. Khomh (Eds.): SSBSE 2024, LNCS 14767, pp. 18–32, 2024.
https://doi.org/10.1007/978-3-031-64573-0_2

achieving high code coverage [16,23] and (2) having fewer smells [22] compared to manually-written test cases, and (3) detecting unknown bugs [1,15].

One of the open challenges in applying MOEAs to test case generation is maintaining a strong selective pressure [19]. Selective pressure is crucial to guide the search towards the Pareto front, where the best solutions are located. The higher the selective pressure, the more likely the algorithm is to find high-quality solutions [19]. However, maintaining selective pressure is challenging when optimizing a large number of objectives as the search space becomes more complex.

One of the key components in MOEAs that increases the selective pressure is the density estimator. Density estimators are used to measure the density/distributions of solutions in the objective space [18]. These estimators introduce innovative methods for comparing and selecting solutions that would otherwise be non-comparable based solely on dominance criteria [18,19]. The *crowding distance* (CD) is a widely used density estimator in MOEAs [18] and the default density estimator used in NSGA-II [8] and by extension DynaMOSA [23].

In this paper, we propose two novel density estimators as an alternative to the classical CD, namely the *token-based density estimator* (TDE) and the *path-based density estimator* (PDE). The TDE and PDE are designed to increase the selective pressure within the domain of test case generation as they measure features that are specific to tests. The *token-based density estimator* measures the semantic distance between test cases using the CodeBERT model tokenizer, *i.e.,* if two tests share similar tokens/keywords. We hypothesize that semantically similar test cases are likely to cover similar parts of the code with similar execution states. Conversely, the *path-based density estimator* considers the distance between the different source-code paths executed by the test cases. We hypothesize that test cases that took a different path through the code to reach a node may result in different internal code states.

To evaluate the proposed density estimators, we conducted an empirical study on 100 non-trivial Java classes from the SF110 benchmark [13,23]. We compared the proposed density estimators with two theoretical state-of-the-art density estimators from the evolutionary computation community for many-objective problems, namely the *subvector-dominance assignment* (SD) and the *epsilon-dominance assignment* (ED) [18], and the classical *crowding distance* (CD) [8] *w.r.t.* their ability to generate test suites with higher mutation score, used as a measure for the fault detection capability.

Our results show that the *path-based density estimator* (PDE) outperforms all other density estimators in enhancing mutation scores. It increases mutation scores by 4.26 % on average (with a max of over 60%) to the traditional crowding distance. The classical crowding distance performed the second best in terms of mutation score. The structural coverage of the different density estimators did not show significant differences.

In summary, we make the following contributions:

1. Two novel density estimators designed for automated test case generation.
2. An empirical evaluation of the proposed density estimators on 100 non-trivial Java classes from the SF110 benchmark.

3. A comparison of the proposed density estimators with two state-of-the-art density estimators from the evolutionary computation community and the classical crowding distance.
4. A full replication package containing the results and the analysis scripts [25].

The structure of the paper is as follows: Section 2 provides background information on many-objective test generation and density estimators. Section 3 describes the proposed density estimators while Sect. 4 describes the experimental setup. Section 5 presents the results of the empirical study, and Sect. 6 discusses the threats to validity. Finally, Sect. 7 concludes the paper and outlines future work.

2 Background and Related Work

Previous research has introduced search-based software test generation (SBST) methods that employ meta-heuristics—and genetic algorithms among others—to create tests at various testing levels, including unit [13], integration [9], and system-level testing [3]. Search-based unit test generation is a particularly active area of study in this field, where iterative optimization algorithms evolve tests towards satisfying multiple criteria (*e.g.,* structural coverage, mutation score) for a given class under test (CUT). Prior research indicates that these techniques effectively achieve high code coverage, enhance fault detection [1], and outperform non-SBST-based approaches [16]. Among others, evolutionary algorithms show better performance than large-language models when generating tests for code not available on GitHub [28] (*i.e.,* the training set) and are not impacted by data leakage issues [26]. SBST techniques have proven successful in testing complex systems [17] and for different programming languages [11,13].

Dynamic Many-Objective Sorting Algorithm (DynaMOSA). The state-of-the-art algorithm for unit test generation is a many-objective evolutionary algorithm called DynaMOSA [23]. Algorithm 1 outlines the pseudo-code of DynaMOSA [23]. This approach targets multiple coverage elements (*e.g.,* lines, branches, mutants) simultaneously as search objectives. To achieve high scalability, DynaMOSA utilizes the hierarchy of dependencies between different coverage targets to update the list of objectives dynamically (lines 5 and 10 in Algorithm 1). The list of objectives is updated at each generation by (1) removing already covered targets and (2) adding new targets that are not covered yet but that are structurally depended on the covered ones. This dynamic approach allows to focus the search on the uncovered targets, thus reducing the search space and improving the search efficiency. Recent independent studies [6,20,24] have shown that DynaMOSA outperforms single-objective and other many-objective evolutionary algorithms *w.r.t.* structural and mutation coverage. Therefore, DynaMOSA currently is the default algorithm in EvoSuite.

Algorithm 1: DynaMOSA

```
   Input:
   U = {u_1, ..., u_m} the set of coverage targets of a program.
   Population size M
   G = ⟨N, E, s⟩: control dependency graph of the program
   φ : E → U: partial map between edges and targets
   Result: A test suite T
1  begin
2      U* ⟵ targets in U with not control dependencies
3      t ⟵ 0 // current generation
4      P_t ⟵ RANDOM-POPULATION(M)
5      archive ⟵ UPDATE-ARCHIVE(P_t, ∅)
6      U* ⟵UPDATE-TARGETS(U*, G, φ)
7      while not (search_budget_consumed) do
8          Q_t ⟵ GENERATE-OFFSPRING(P_t)
9          archive ⟵ UPDATE-ARCHIVE(Q_t, archive)
10         U* ⟵UPDATE-TARGETS(U*, G, φ)
11         R_t ⟵ P_t ⋃ Q_t
12         𝔽 ⟵ PREFERENCE-SORTING(R_t, U*)
13         P_{t+1} ⟵ ∅
14         d ⟵ 0
15         while | P_{t+1} | + | 𝔽_d | ≤ M do
16             CROWDING-DISTANCE-ASSIGNMENT(𝔽_d, U*)
17             P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d
18             d ⟵ d + 1
19         Sort(𝔽_d) //according to the crowding distance
20         P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d[1 : (M− | P_{t+1} |)]
21         t ⟵ t + 1
22     T ⟵ archive
```

2.1 Density Estimator for Many-Objective Optimization

A key feature in DynaMOSA, as in any many-objective algorithms, is the density-estimation method used to increase the selection pressure towards the Pareto front. Selective pressure is achieved by (1) sorting the test cases in the population based on the *preference criterion* (line 12 in Algorithm 1) and (2) selecting the best tests based on their *crowding distance* (line 16 in Algorithm 1). The former promotes the test cases closer to each objective (coverage target), and the latter promotes the test cases more diverse in the objective space. More specifically, the crowding distance is calculated as the sum of the differences in the objective values of the two neighboring test cases [8]. However, as pointed out by Köppen and Yoshida [18], the crowding distance does not scale well with the number of objectives, as it may assign the maximum distance (infinite) to all test cases in the first non-dominated front. To address this limitation, they proposed alternative methods to calculate the crowding distance, such as the *sub-vector dominance* and *epsilon dominance assignment*. We elaborate on these methods in the following sections.

Sub-vector Dominance Assignment. The first alternative estimator introduced by Köppen and Yoshida [18] for many-objective numerical problems is the *sub-vector dominance*. This estimator is applied to each non-dominated front, and therefore, it is used to calculate the solution density for all solutions (test cases in our context) within the same front. The algorithm first assigns an

infinite distance to single-member fronts, suggesting that an individual inherently exhibits maximum diversity. For fronts containing multiple solutions, it sets each test case's distance to the highest possible value, indicating that their dominance still needs to be evaluated. Then, this estimator processes each pair of test cases within the same front. For every pair (`p1`, `p2`), it assesses their effectiveness against each objective in the set using the corresponding objective values/scores. Two counters `dominate1` and `dominate2` are used to count the number of objectives where `p1` performs better than `p2` and vice versa. These counters determine how much (the strength) each test case is dominated by the other across all objectives.

After evaluating the dominance of the test cases, the distance for each test case is adjusted. The new distance is the smaller value between its current distance and the number of goals where it is found to be inferior. Essentially, this means that the more a test case is dominated by others, the smaller its distance becomes, reflecting its relative performance deficit across the objectives. Hence, this estimator favors test cases demonstrating superior performance over a broader range of objectives, fostering a varied pool of solutions throughout the evolutionary cycle.

Epsilon-Dominace Assignment. The epsilon-dominance assignment provides a more detailed comparison between solutions than sub-vector dominance. Instead of simply counting how many objectives a solution falls short on, it measures how much worse a solution is in each objective. Similarly, to the other density estimators, this method is applied to each non-dominated front.

For each non-dominated solution `p1`, this estimators first calculates all ϵ-dominance scores of `p1` compared to all other solutions in the same fronts. For each pair of solutions (`p1`, `p2`), this metric considers all objective values of `p2` that are worse than the corresponding objectives of `p1`. The epsilon dominance of `p1` over `p2` is the smallest value ϵ that, if subtracted from all objectives of `p2`, makes `p2` Pareto-dominating `p1`. This concept is often called additive ϵ-dominance in the related literature [18]. Finally, the density measure of a solution is computed as the smallest of all its epsilon dominance calculated *w.r.t.* to all other solutions in the same non-dominated front. The larger the distance for a solution `p1`, the higher the "effort" or the epsilon value needed to make the other solutions dominate `p1`.

3 Density Estimators for Test Cases

In this section, we present two novel density estimators for test cases as alternative to the crowding distance, i.e., line 16 of Algorithm 1. These estimators are designed to measure (and thus promote) the diversity of the test cases generated by DynaMOSA. The first estimator is based on the semantic content of the test cases (or *genotype*) related to the keywords/tokens that form the test cases. The second estimator works in the objective space (also called *phenotype*) and measures the diversity of the execution paths covered by the test cases. We

describe the two estimators in detail and discuss their integration into the main loop of the DynaMOSA algorithm.

3.1 Token-Based Distance Assignment

We introduce this novel distance assignment with the idea of using natural language processing (NLP) methods to measure the *semantic diversity* of test cases. Our intuition is that at the same level of dominance, test cases that are semantically more diverse should be assigned a higher probability of mating since they may cover different paths in the CUT or use more diverse input values.

To measure the semantic content of the test cases, we rely on the tokenizers of large language models (LLMs), particularly the CodeBERT pre-trained model by Microsoft [12] and publicly available on `HuggingFace`[1]. The CodeBERT tokenizer is designed to understand both programming and natural languages as it operates similarly to the tokenization process of BERT model [10] but with customization to handle code syntax and semantics.

CodeBERT uses the Byte-Pair Encoding (BPE) algorithm for its tokenization [12], which combines both character- and word-level tokenization. BPE builds an *initial vocabulary* of individual characters and gradually builds up a vocabulary of more frequent and longer sub-word units (byte pairs) by combining pairs of symbols (or characters) that frequently occur together. BPE iteratively counts the frequency of pairs of adjacent symbols in the corpus and merges the most frequent pair to create a new symbol (*iterative learning*). This process is repeated for a predefined number of merge operations, leading to a final vocabulary that includes a mix of characters, common sub-words, and full words. The *tokenization* of new text (test cases in our case) is applied by splitting it into individual characters and then applying the merge rules learned during training. The merging procedure combines characters and sub-words into the tokens present in its final vocabulary.

At the end of the tokenization process, each test case (here considered as text) is tokenized into different tokens, grouped in *special* and *non-special* tokens. The former tokens are essential for the model to understand code structure. For instance, the `[SEP]` tokens delimit different segments within the input sequence, such as demarcating the end of a code snippet and the beginning of a natural language comment or vice versa. Instead, non-special tokens are the regular tokens representing the input text's content (tests in our case).

Token-Based Density Estimator. Algorithm 2 outlines the pseudo-code of the distance assignment metric based on the token frequency. The algorithm takes in input the current set of objectives U^*, and a list of non-dominated test cases $\mathbb{F}_i$. The algorithm starts by initializing two maps: (1) a mapping of test cases to their respective token sets (`TokenMap` in line 2), and (2) a mapping of

[1] https://huggingface.co/microsoft/codebert-base.

Algorithm 2: Token-based Distance Assignment

```
   Input:
   U*: current set of objectives
   F_i list of non-dominated test cases
   Output: Updated test cases with distance values
1  begin
2      TokenMap ⟵ empty map                                  // mapping test cases to tokens
3      TokenFrequency ⟵ empty map                  // mapping tokens to their frequencies
4      foreach τ ∈ F_i do
5          text ⟵ TO-TEXT(τ)
6          tokens ⟵ TOKENIZER(text)                          // Applying CodeBERT tokenizers
7          TokenMap[τ] ⟵ tokens
8          /* Update tokens frequencies                                                  */
9          foreach token ∈ tokens do
10             if token ∈ TokenFrequency then
11                 TokenFrequency[token] ⟵ TokenFrequency[token] + 1
12             else
13                 TokenFrequency[token] ⟵ 1
14             end
15         end
16     end
17     foreach τ ∈ F_i do
18         tokens ⟵ TokenMap[τ]
19         frequency ⟵ ∞
20         foreach token ∈ tokens do
21             frequency ⟵ MIN(TokenFrequency[token], frequency)
22         end
23         SET-DIVERSITY(τ) = 1.0 / frequency
24     end
25 end
```

tokens to their occurrence frequencies across all test cases (`TokenFrequency` in line 3). Then, the algorithm tokenizes the test and updates the token frequencies among all test cases.

Each test case τ is converted into its list of tokens using the CodeBERT tokenizer (function TOKENIZER in line 6). The resulting tokens are stored in the `TokenMap` and associated with τ in the mapping. Subsequently, the algorithm updates token frequencies stored in `TokenFrequency` by iterating over each token in the tokens set of τ. The token frequencies calculated in lines 9–14 of Algorithm 2 are used to compute a diversity value for each test case with the loop in lines 17–24. Specifically, the algorithm calculates the minimum token frequency for all tokens of a test case τ (lines 19–22). Finally, the assigned distance for τ is calculated as the inverse of this minimum token frequency (line 23).

This token-based metric prioritizes test cases containing rarer tokens, assuming such tests may explore paths or scenarios in the software under test that are less frequently executed. We rely on the CodeBERT tokenizer as it allows us to capture nuances in the code that textual-based methods might miss.

3.2 Path-Based Density Estimator

We proposed a new substitute distance assignment tailored for test case generation and based on dynamic information from the test execution results. Our intuition is that test cases that reach the coverage frontier (i.e., the yet uncov-

Algorithm 3: Path-based Distance Assignment

```
   Input:
   U*: current set of objectives
   F_i list of non-dominated test cases
   Output: Updated test cases with distance values
1  begin
2      foreach τ_i ∈ F_i do
3          L_i = COVERED-LINES(τ_i)                          // set of lines covered by τ_i
4          foreach τ_j ∈ F_i do
5              L_j = COVERED-LINES(τ_j)                      // set of lines covered by τ_j
6              distances(τ_i, τ_j) ⟵ JACCARD-DISTANCE(L_i, L_j)
7              distances(τ_j, τ_i) ⟵ distance(τ_i, τ_j)
8          end
9      end
10     visited ⟵ ∅                                          // Set of already-visited test cases
11     for index ← 0 to | F_i | do
12         bestTest ⟵ ∅                                     // test case to select
13         maxDiversity ⟵ −∞                                // diversity of the test case to select
14         foreach τ ∈ F_i do
15             if τ ∉ visited then
16                 distance ⟵ AVERAGE-DISTANCE(index, visited, distances)
17                 /* Select the case with the largest distance to the already considered
                      ones                                                              */
18                 if distance ¿ maxDiversity then
19                     maxDiversity ⟵ distance
20                     bestTest ⟵ τ
21                 end
22             end
23         end
24         visited ⟵ visited + {τ}
25         SET-DIVERSITY(τ) = maxDiversity
26     end
27 end
```

ered targets/branches) passing through different/diverse execution paths of the software under test are more likely to lead to more diverse execution states (e.g., class attributes and internal variable values).

The new assignment procedure is outlined in Algorithm 3. It leverages line coverage data from previously executed tests in DynaMOSA's early stages, thus avoiding re-execution. For any two test cases, τ_i and τ_j, we compute their *Jaccard distance* based on the sets of lines covered by each, as follows:

$$Jaccard(\tau_i, \tau_j) = 1 - \frac{\mid L_i \cap L_j \mid}{\mid L_i \cup L_j \mid} \tag{1}$$

where L_i and L_j represent the lines covered by τ_i and τ_j, respectively. This metric quantifies the dissimilarity in code coverage between test cases, accounting for all lines covered during execution, including those outside the class under test (e.g., the lines covered for input objects). The pairwise distances are stored in the `distances` matrix in lines 6–7 of Algorithm 3.

Subsequently, our algorithm adopts a greedy strategy to select test cases that maximize diversity iteratively (lines 11–24 in Algorithm 3). Initially, it selects the test case with the highest Jaccard distance from the pre-computed distances

matrix `distances`. The selected test is added to the set of visited tests (`visited`) and assigned a distance equal to its maximum Jaccard distance. In each following iteration, the greedy strategy calculates the average Jaccard distance for all test cases that have not been selected yet (if-condition in line 15) using the AVERAGE-DISTANCE function (line 16). This function calculates the average distance of a test case τ to all other previously chosen ones and stored in `visited`. Among the yet-to-select test cases, the algorithm greedily chooses the one with the largest average Jaccard distance (lines 18–21). The diversity of the selected test case (`bestTest`) is then updated to reflect this maximum value (line 25); it is marked as visited (line 24). This process repeats until all test cases are selected and assigned a diversity value, reflecting their contribution to covering diverse execution paths within the software under test.

Code Optimization. To speed up the calculation of the pairwise distances (for our estimators), we pre-allocate a square matrix to store the distances between all test cases, whose dimension (number of columns/rows) is equal to the size of the front. In the worst-case scenario, the font size corresponds to the population size. However, DynaMOSA can increase the population size if, during the preference criterion calculation, the first front is larger than the population size. In this case, the population size is increased to the first font size, which requires increasing the size of the distance matrix. In case the population size is smaller than the matrix dimension, the latter is not scaled but kept at the largest values in case the population size increases again in subsequent search iterations.

Pre-allocating a matrix of fixed size was critical to (1) speed up the search, as allocating many large matrices incurs a high computational cost, and (2) avoid the overhead of dynamically resizing the matrix during the search. This is also critical to avoid memory exhaustion since creating a new matrix for each iteration will consume more significant memory and at a pace that is too fast for the garbage collector to free the memory. We did experience indeed many memory-related crushes and issues when we did not pre-allocate the matrix.

4 Empirical Study

To investigate the performance of the proposed density estimators within the context of test case generation, we perform an empirical evaluation to answer the following research question:

RQ *How do the proposed density estimators compare to the classical crowding distance w.r.t. mutation score?*

More specifically, we look at the performance of (i) two state-of-the-art density estimators from the evolutionary computation community, namely the *subvector-dominance assignment* (SD) and the *epsilon-dominance assignment* (ED), when applied to the context of test case generation and (ii) two novel density estimators created specifically for test case generation introduced in this work, namely the *token-based density estimator* (TDE) and the *path-based density estimator* (PDE).

4.1 Benchmark

We performed the evaluation on a subset of the SF110 benchmark [14], which is a widely used benchmark in the literature for evaluating test case generation techniques for Java [14,21,23,24]. We do not consider the whole SF110 corpus as many classes are trivial [27] and the total number of classes in the corpus (23,886 Java classes) would take too long to run. Specifically, we randomly selected 100 classes from the SF110 corpus with non-trivial complexity (Cyclomatic Complexity (CC) > 3). This same selection procedure has been used in related literature [21,23,24].

4.2 Parameter Settings

For the parameter settings, we adopted the defaults used by EvoSuite [13] (test case generation tool used in our experiment). These settings have been widely used in literature and previous studies have shown that although parameter tuning impacts the performance of search algorithms, the default parameter values provide reasonable and acceptable results [5]. Therefore, we used DynaMOSA [23] using a single point crossover with a crossover probability of 0.75, mutation with a probability of $1/n$ (n = number of statements in the test case), tournament selection, and a population size of 50. As we are focussing on fault detection, we set *branches* and *strong mutation* as the objectives to optimize. The search budget per unit under test is 300 s.

4.3 Experimental Protocol

To answer the research question, we ran EvoSuite with the four density estimators (SD, ED, TDE, and PDE) and the crowding distance (CD) as a baseline on the 100 classes from the SF110 benchmark and recorded the final branch coverage and mutation score achieved by the generated test cases. To account for the stochastic nature of search-based test case generation, each unit under test was run 20 times. In total, we performed 10 000 runs, consisting of 20 repetitions of 5 configurations on 100 units under test. This required $(10000\ \text{runs} \times 300\,\text{s})/(60\,\text{s} \times 60\,\text{min} \times 24\,\text{h}) \approx 35\,\text{d}$ of consecutive computation time. The experiment was performed on a system with an AMD Ryzen Threadripper PRO 3995WX (64 cores 2.7 GHz) with 256 GB of RAM.

After the experiment, we compared the mutation score achieved by the test cases generated using the different density estimators and performed statistical analysis. We applied the unpaired Wilcoxon signed-rank test [7] with a threshold of 0.05. This non-parametric statistical test determines if two data distributions are significantly different enough to reject the null hypothesis that the two distributions are equal. In addition, we apply the Vargha-Delaney $\hat{A}_{12}$ statistic [29] to determine the effect size of the result, which determines the magnitude of the difference between the two data distributions.

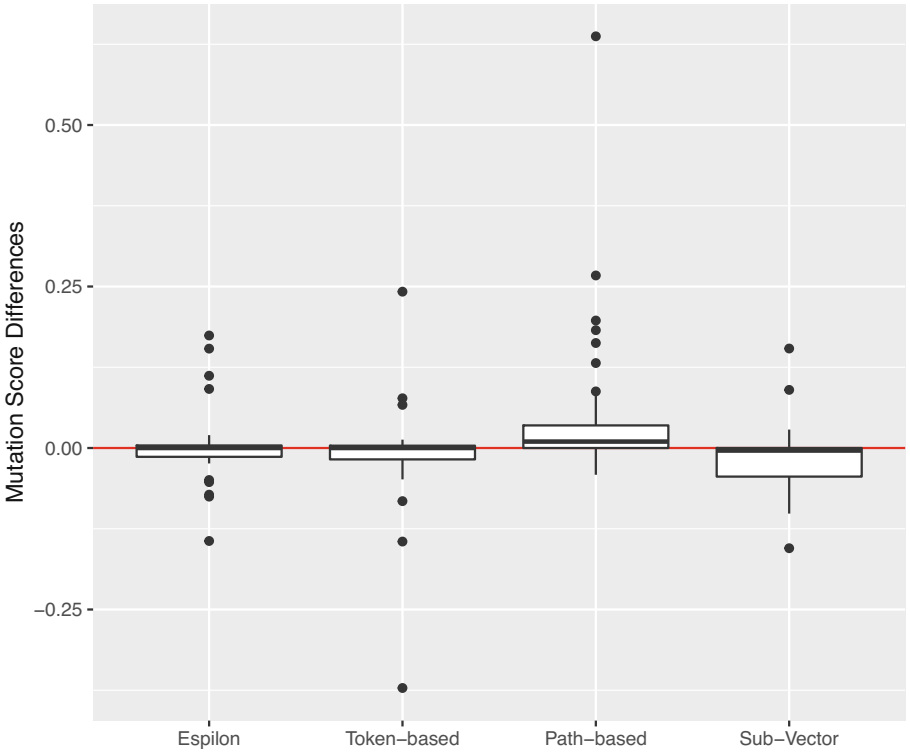

Fig. 1. Difference in achieved mutation score across the classes in the benchmark using the different density estimators compared to the crowding distance

5 Results

This section presents the results of our empirical study. All differences in results are presented in absolute differences (percentage points).

Figure 1 shows the difference in the mutation score achieved by the test cases generated using the different density estimators compared to the classical crowding distance. The datapoints in the boxplot represent the difference in the median mutation score for each class in the benchmark. The results show that the *path-based density estimator* (PDE) achieves the highest mean mutation score (53.90 %) across the classes in the benchmark and improves the most over crowding distance. The crowding distance (CD) achieves a mean mutation score of 49.64 %. The *token-based density estimator* (TDE) has a mean mutation score of 49.17 %, which is slightly lower than the crowding distance. The *epsilon-dominance assignment* (ED) and the *subvector-dominance assignment* (SD) achieve mean mutation scores of 48.83 % and 48.64 %, respectively.

We, additionally, performed a statistical analysis to determine the significance of the differences in the mutation score achieved by the test cases generated using the different density estimators. Table 1 shows the results of this

Table 1. Results of the statistical analysis of the achieved mutation score using Vargha-Delaney $\hat{A}_{12}$ statistic

Comparison	#Win			#No diff.	#Lose		
	Large	Medium	Small	Negl	Small	Medium	Large
Path-based vs. Token-based	18	11	4	63	-	2	2
Path-based vs. Epsilon	16	10	7	62	1	3	1
Path-based vs. Sub-Vector	17	12	2	66	-	3	-
Path-based vs. Crowding	12	12	3	71	1	1	-
Token-based vs. Epsilon	2	7	3	73	3	8	4
Token-based vs. Sub-Vector	7	8	-	76	1	5	3
Token-based vs. Crowding	2	2	-	85	2	4	5
Epsilon vs. Sub-Vector	3	2	2	90	-	2	1
Epsilon vs. Crowding	4	1	4	79	3	6	3
Sub-Vector vs. Crowding	3	2	-	82	1	5	7

statistical analysis based on a p-value ≤ 0.05. In this table, the *#Win* columns indicate the number of times that the left density estimator has a statistically significant improvement over the right one, the *#No diff.* column indicates the number of times that there is no evidence that the two competing density estimators are different, and the *#Lose* columns indicate the number of times that the left density estimator has statistically worse results than the right one. The *#Win* and *#Lose* columns also include the $\hat{A}_{12}$ effect size, classified into *Small*, *Medium*, and *Large*.

The results show that the *path-based density estimator* (PDE) outperforms the other density estimators in most comparisons. In particular, PDE outperforms the *epsilon-dominance assignment* (ED) in 33 out of 100 comparisons, the *token-based density estimator* (TDE) in 33 out of 100 comparisons, the *subvector-dominance assignment* (SD) in 31 out of 100 comparisons, and the *crowding distance* (CD) in 27 out of 100 comparisons. The *epsilon-dominance assignment* (ED) outperforms the *token-based density estimator* (TDE) in 15 out of 100 comparisons, the *subvector-dominance assignment* (SD) in 7 out of 100 comparisons, and the *crowding distance* (CD) in 9 out of 100 comparisons. The *token-based density estimator* (TDE) outperforms the *subvector-dominance assignment* (Sub-Vector) in 15 out of 100 comparisons and the *crowding distance* (CD) in 4 out of 100 comparisons. Lastly, the *subvector-dominance assignment* (SD) outperforms the *crowding distance* (CD) in 5 out of 100 comparisons. Interestingly, the classical crowding distance (CD) performs better in more cases than the *subvector-dominance assignment* (SD), the *epsilon-dominance assignment* (ED), and the *token-based density estimator* (TDE). However, in the majority of the classes in the benchmarks, there is no significant difference between the density estimators.

In addition to the mutation score, we also looked at the branch coverage achieved by the generated test cases. We observed that the branch coverage achieved by the test cases generated using the different density estimators is identical. This indicates that the difference in mutation score is not due to differences in branch coverage but rather due to the improvement in the density estimators.

6 Threats to Validity

This section discusses the potential threats to the validity of our study.

External Validity: One of the threats to the *external validity* of our study is the selection of the benchmark. The selection of the benchmark impacts the generalizability of the results. To address this threat, we used a subset of the SF110 benchmark, which is a widely used benchmark in the literature for evaluating test case generation techniques for Java. The subset of the benchmark was selected based on the complexity of the classes to ensure that the results are not biased by trivial classes. However, the results may not generalize to other benchmarks or programming languages.

Conclusion Validity: The stochastic nature of search-based test case generation introduces a threat to the *conclusion validity* of our study. To mitigate this threat, we ran each configuration 20 times with different random seeds. This allows us to draw statistically significant conclusions from the results. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [4]. Additionally, we used the unpaired Wilcoxon signed-rank test and the Vargha-Delaney $\hat{A}_{12}$ effect size to assess the significance and magnitude of our results.

7 Conclusions and Future Work

In this paper, we have presented two novel density estimators for automated test case generation to increase the selective pressure within the search front. We compared the proposed density estimators with two state-of-the-art density estimators from the evolutionary computation community and the classical crowding distance. Our results show that our proposed *path-based density estimator* (PDE) is the most effective in promoting the diversity of the solutions in the population, leading to a better spread of the solutions in the objective space and a higher mutation score. The classical crowding distance performed the second best in terms of mutation score.

In future work, we will evaluate the proposed density estimators on other test generation problem—*e.g.,* system-level test case generation—and other software testing problems, diversity-based test case prioritization. We also plan to (1) use different tokenizers as well as (2) different LLMs for the test case embeddings as alternatives to CodeBERT. Finally, we plan to analyze the relation between the diversity of the test cases and the fault detection capability of the generated test suites.

References

1. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: 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), May 2017, pp. 263–272. IEEE (2017). https://doi.org/10.1109/ICSE-SEIP.2017.27
2. Arcuri, A.: Test suite generation with the many independent objective (MIO) algorithm. Inf. Softw. Technol. **104**, 195–206 (2018)
3. Arcuri, A.: RESTful API automated test case generation with EvoMaster. ACM Trans. Softw. Eng. Methodol. (TOSEM) **28**(1), 1–37 (2019)
4. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verification Reliab. **24**(3), 219–250 (2014)
5. Arcuri, A., Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. Empir. Softw. Eng. **18**, 594–623 (2013)
6. Campos, J., Ge, Y., Albunian, N., Fraser, G., Eler, M., Arcuri, A.: An empirical evaluation of evolutionary algorithms for unit test suite generation. Inf. Softw. Technol. **104**, 207–235 (2018). https://doi.org/10.1016/j.infsof.2018.08.010
7. Conover, W.J.: Practical Nonparametric Statistics, vol. 350. Wiley, Hoboken (1999)
8. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
9. Derakhshanfar, P., Devroey, X., Panichella, A., Zaidman, A., van Deursen, A.: Towards integration-level test case generation using call site information. arXiv preprint arXiv:2001.04221 (2020)
10. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
11. Erni, N., Mohammed, A.A.M.A., Birchler, C., Derakhshanfar, P., Lukasczyk, S., Panichella, S.: SBFT tool competition 2024–Python test case generation track. arXiv preprint arXiv:2401.15189 (2024)
12. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
13. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416–419 (2011)
14. Fraser, G., Arcuri, A.: A large-scale evaluation of automated unit test generation using EvoSuite. ACM Trans. Softw. Eng. Methodol. (TOSEM) **24**(2), 1–42 (2014)
15. Fraser, G., Arcuri, A.: 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. Empir. Softw. Eng. **20**, 611–639 (2015)
16. Jahangirova, G., Terragni, V.: SBFT tool competition 2023-Java test case generation track. In: 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), pp. 61–64. IEEE (2023)
17. Khatiri, S., Saurabh, P., Zimmermann, T., Munasinghe, C., Birchler, C., Panichella, S.: SBFT tool competition 2024: CPS-UAV test case generation track. In: 17th International Workshop on Search-Based and Fuzz Testing (SBFT), Lisbon, Portugal, 14–20 April 2024. ZHAW Zürcher Hochschule für Angewandte Wissenschaften (2024)
18. Köppen, M., Yoshida, K.: Substitute distance assignments in NSGA-II for handling many-objective optimization problems. In: Obayashi, S., Deb, K., Poloni,

C., Hiroyasu, T., Murata, T. (eds.) EMO 2007. LNCS, vol. 4403, pp. 727–741. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70928-2_55
19. Li, B., Li, J., Tang, K., Yao, X.: Many-objective evolutionary algorithms: a survey. ACM Comput. Surv. (CSUR) **48**(1), 1–35 (2015)
20. Lukasczyk, S., Kroiß, F., Fraser, G.: An empirical study of automated unit test generation for Python. Empir. Softw. Eng. **28**(2), 36 (2023)
21. Molina, U.R., Kifetew, F., Panichella, A.: Java unit testing tool competition: sixth round. In: Proceedings of the 11th International Workshop on Search-Based Software Testing, pp. 22–29 (2018)
22. Panichella, A., Panichella, S., Fraser, G., Sawant, A.A., Hellendoorn, V.: Test smells 20 years later: detectability, validity, and reliability. Empir. Softw. Eng. **27**, 170 (2022). https://doi.org/10.1007/s10664-022-10207-5
23. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Trans. Softw. Eng. **44**(2), 122–158 (2017)
24. Panichella, A., Kifetew, F.M., Tonella, P.: A large scale empirical comparison of state-of-the-art search-based test case generators. Inf. Softw. Technol. **104**, 236–256 (2018)
25. Panichella, A., Mitchell, O.: Replication package of "higher fault detection through novel density estimators in unit test generation", May 2024. https://doi.org/10.5281/zenodo.11209898
26. Sallou, J., Durieux, T., Panichella, A.: Breaking the silence: the threats of using LLMs in software engineering. In: ACM/IEEE 46th International Conference on Software Engineering. ACM/IEEE (2024)
27. Shamshiri, S., Rojas, J.M., Fraser, G., McMinn, P.: Random or genetic algorithm search for object-oriented test suite generation? In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1367–1374 (2015)
28. Siddiq, M.L., Santos, J., Tanvir, R.H., Ulfat, N., Rifat, F.A., Lopes, V.C.: Exploring the effectiveness of large language models in generating unit tests. arXiv preprint arXiv:2305.00418 (2023)
29. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000)