An Evaluation of Distance Based Test Suite Reduction Techniques

Alessandro Escher
Technical University of Munich
Munich, Germany
alessandro.escher@tum.de

Raphael Nömmer
Technical University of Munich
Munich, Germany
noemmer@cqse.eu

***Abstract*—Efficient test suite selection is crucial in software testing due to the high cost of running extensive tests, particularly on large industry projects. Coverage-based techniques aim to maximize system execution within time constraints but often suffer from costly and complex coverage recording processes. This study explores alternative selection methods using test metadata and source code. Hierarchical Agglomerative Clustering (HAC) and a greedy approach were evaluated alongside distance measures based on package path distance and vector representations of test code.**

Evaluation on a variety of open-source projects and a large industry project revealed that while the proposed methods maintained decent coverage, they did not significantly outperform a strictly time-based selection. We note that HAC lacks a clear time-budget stopping criterion and performs worse than the greedy approach and random selection. Furthermore, techniques that rely on execution times tend to neglect longer-running tests, which can have an impact on fault detection, particularly in industry projects.

This study emphasizes the importance of effective test selection methods that balance coverage, cost, and fault detection. We suggest that a simple yet effective baseline such as lowest execution time first is a more robust baseline than a random selection, especially for a cost based evaluation, and underline the need for more competitive baseline methods in test suite optimization research.

***Index Terms*—test selection, test suite reduction, clustering, code embeddings, topic model**

I. Introduction

Software testing is an integral part of the software development lifecycle of any application. In order to validate that the program works as intended and provides the required functionality, a suite of tests is run—each focusing on different components of the system and at differing granularities—at various points in time before the software is released. Regression testing is a popular approach for this. The test suite is run at different intervals, depending on the size of the suite and requirements of the project. Most often this is done whenever a change is made to the system as this is typically where faults are introduced [1]. For large industry systems where test suites can reach hours or days of execution time, this takes up a significant amount of resources [2]–[6], causing additional costs for the company and resulting in slower feedback for the developers. Test Case Selection (TCS) aims to alleviate these issues by selecting a subset of the test suite, picking relevant tests and omitting redundant ones. Many TCS approaches rely on the test coverage—be that at the statement, branch or method level—of the test suite in order to determine which tests to choose. Recording and storing this coverage data can become a cumbersome process, especially for large and complex software systems that use multiple programming languages and frameworks [7]. Because of this, a company will have to struggle with the high cost and maintenance effort, and may only decide to do adopt this approach in a limited manner [8]. Being able to use an alternative approach that is not based on coverage data but instead uses readily available data would allow for TCS to be performed on all projects, no matter their priority. Additionally, it would allow the developers of a project to gain immediate benefits of TCS in case the coverage recording process is not set up yet.

In this study we focus on exploring alternative approaches to coverage-based test suite selection, aiming to address the challenges associated with the expense and complexity of traditional methods. Specifically, we investigate the feasibility of using test metadata and source code for a more efficient test selection. We examine a clustering and a greedy approach in conjunction with various distance measures based on package path distance and vector representations of test code. The practical effectiveness of these techniques in maintaining coverage and detecting faults is evaluated across a variety of open source projects as well as a large industry project.

The rest of this research is structured as follows. Section II gives background information about some of the techniques and concepts used. In Section III, we explain our TCS approaches and the different combination of parameters that we apply. Afterwards in Section IV we detail our empirical evaluation of our proposed implementation and lastly, we offer our concluding thoughts in Section V.

II. Related Work & Background

This section gives background information about the concept of test selection and some of the techniques that were used and offers insight into how they have been applied in related works.

A. Test Suite Optimization

Optimizing a test suite entails maximizing its effectiveness, that is its achieved coverage and fault detection for a given cost in execution time [2]. There are different principles that

have been suggested for doing this, which include permanently or temporarily removing some of the tests or changing the order in which they are executed. There are three different techniques often discussed in test optimization literature.

Test Suite Minimization (TSM): Since test suites increase over time for growing systems, and tests are rarely removed from the code base, the number of redundant or similar tests can accumulate over time, resulting in longer execution times and slower feedback to the developer [9]. TSM, with the goal of permanently removing redundant tests from a test suite, is performed occasionally when the test suite has grown significantly [9].

Test Case Prioritization (TCP): Tests can also be ordered so failing tests are executed sooner and the developer gets quicker feedback on whether their changes introduced any faults into the system. This technique is called TCP and has found a lot of interest in research [10]–[12]. The advantage of TCP is that the whole test suite is always run, such that there can be no loss in effectiveness. In practice, this is often times not enough as test suites of large industry projects can reach days of execution times, causing them to be run only in longer intervals [8].

Test Case Selection: The basic goal of TCS is to reduce the size of a test suite by selecting only a subset of tests to run, while still maintaining the test suite's ability to detect defects. This process is not permanent and can be done on the fly at different intervals and at different stages of the Continuous Integration (CI) process. TCS is often used in the context of regression testing, for example after a change has been made to the code base. In this case, it is important for the developer to get quick feedback on whether their changes introduced any faults in the system. In a literature survey performed by Kazmi et al. it was found that 70% of proposed TCS strategies focus on the costs of the test suite as an evaluation measure, while 31% focus on fault detection as a measure and only 16% use coverage [2].

In this work we establish and evaluate several distance based TCS strategies, which additionally use the execution time as a cost measure, compare them to a coverage based approach and evaluate them based on their coverage and failure detection for a given cost.

B. Clustering

The general aim of clustering is to group elements together based on some definition of similarity, such that elements in a group are similar to each other, while elements from different groups are dissimilar to each other. In this paper we focused on hierarchical clustering techniques, which is a deterministic clustering approach that outputs a hierarchy of clusters [13]. Hierarchical clustering can be further divided into Hierarchical Agglomerative Clustering (HAC) and Divisive Clustering, which differ in the way that clusters are created. Divisive Clustering starts with one cluster containing all elements and iteratively splits clusters into subclusters. HAC, on the other hand, starts with each element in its own cluster and then iteratively merges the two most similar clusters into one.

HAC has found numerous applications in test selection. Sapna and Mohanty used HAC to select use case diagrams in the form of scenarios and find that it performs better than random selection [14]. Coviello et al. perform test suite reduction using HAC on a set of open source Java systems and found that they were able to greatly reduce the size of the test suite while incurring an acceptable loss in failure detection [15]. Carlson et al. performed their evaluation on an industrial system at *Microsoft* and found that employing a clustering based test selection under time constraints improved the failure detection when compared to a selection without clustering [11].

In this work we employ HAC as one of our selection strategies and combine it with various distance metrics. We evaluate it on both an industry and open source projects and compare it to other selection strategies based on the retained coverage of the selected test suite.

C. Code Language Models

Understanding and interpreting source code has been greatly enhanced by recent advancements in language models. The introduction of the Transformer model architecture by Vaswani et al. [16] and the subsequent works derived from it constituted a great enhancement in Natural Language Processing (NLP) tasks. The Transformer allows for a more efficient training process while surpassing the performance of previous recurrent or convolutional architectures [16]. Combined with additional advances in the form of pre-trained deep neural networks—originally aimed at NLP tasks [17]—a wide range of Large Language Models (LLMs) have been derived in the last years.

Concepts originally applied to NLP tasks [17] have since also been used for so called *code intelligence* tasks. *CodeBERT* is a model that is able to combine natural language instructions and programming language, allowing it to excel at downstream tasks such as code search and code comment generation [18]. *GraphCodeBERT* improves on this by additionally considering data flow in its pre-training process [19]. Newer models such as *UniXcoder* [20] and *CodeT5+* [21] use modern enhancements to the Transformer architecture and aim to provide more flexibility regarding the downstream tasks, such as code completion, code generation and clone detection, that the model can be applied to.

In this work, we leverage the ability to understand and interpret source code that these models have aquired in their pre-training, and use it to obtain a vector representation for a test's source code.

III. Implementation

This section describes the different TCS approaches that we implemented and their underlying distance metrics.

A. Selection Strategy

The primary goal of a TCS strategy is to reduce a given test suite by selecting a subset of its tests such that the selected tests still adequately detect failures. The selection is

performed until some stopping criteria is reached, depending on the TCS. Stopping criteria include the number of selected tests—selection stops once the specified amount of tests has been selected, or the execution time of the reduced test suite—selection stops once the selected tests exhaust the chosen testing budget. In this paper, we implemented and evaluated two different TCS strategies.

1) Agglomerative Clustering: HAC (see Section II-B) is used in order to group tests into clusters with the goal of achieving a high similarity between elements inside of a cluster and low similarity between elements in different clusters. The average link criterion is used in order to determine the distance between two clusters, which takes into account the mean of the distances of all elements between both clusters [22]. Once the clustering process is stopped, we pick the test with the lowest execution time from each cluster. The resulting set of test cases constitutes the selected test suite.

2) Greedy Selection: The greedy selection strategy aims at iteratively picking the test that has the largest distance score from the already selected ones, as described in Algorithm 1. At first, the quickest test is determined (ties are broken up randomly) and added to the initially empty set of selected tests *S*, while all other tests are placed into the set of remaining tests *R*. In each iteration, the mean distance from each remaining test to all selected tests is calculated using a distance metric as described in Section III-B. This distance is then divided by the candidate's execution time (measured in milliseconds) to obtain a final distance score. The candidate with the largest distance score is added to the set of selected tests. This process is continued until one of the stopping criteria is reached or until there are no more remaining tests. In the latter case, the selection strategy picked all tests and no reduction was performed.

Algorithm 1 Greedy Test Selection

```
1:  procedure GreedySelection(Tests, StoppingCriteria)
2:      S ← ∅                                          ▷ Selected tests
3:      R ← Tests                                      ▷ Remaining tests
4:      t ← selectFastest(R)
5:      S ← S ∪ {t}                                    ▷ Add t to S
6:      R ← R \ {t}                                    ▷ Remove t from R
7:      while ¬ StoppingCriteria and R ≠ ∅ do
8:          maxDistanceScore ← −∞
9:          selectedCandidate ← null
10:         for candidate in R do
11:             distance ← MeanDistance(S, candidate)
12:             distanceScore ← distance / (candidate.executionTime+1)
13:             if distanceScore > maxDistanceScore then
14:                 maxDistanceScore ← distanceScore
15:                 selectedCandidate ← candidate
16:         S ← S ∪ {selectedCandidate}
17:         R ← R \ {selectedCandidate}
18:         Update StoppingCriteria                    ▷ Based on selectedCandidate
19:     return S
```

B. Distance Metric

To quantify the distance—or similarity—of two test cases, a distance metric is required that takes two test cases as input and outputs a numeric value. In this case, a higher value indicates a greater distance between the two elements. One of the main goals of this paper is to investigate metrics that do not rely on coverage data, but instead on readily available data such as the source code of a test or its package path.

1) Package Path: The package path distance metric is a very simple way of denoting the distance of two test cases by considering their location in the system's package namespace. The package namespace is represented as a tree structure, where the test cases are the leaf nodes. To compute the distance between two test cases, the common ancestor of both tests is determined and then the number of edges that lie on the path from one leaf to the other via the common ancestor is used as the distance value. For example, *org.example.service.A* and *org.example.core.B* have a path distance of 4 via the common ancestor *org.example*.

2) Vector Based: By describing a test case as a numerical vector, existing vector distance metrics such as the Euclidean distance (1) or the Cosine dissimilarity (2) can be used to compute a distance value.

$$\text{Euclidean}(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2} \tag{1}$$

$$\text{Cosine}(\mathbf{u}, \mathbf{v}) = 1 - \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} \tag{2}$$

One such vectorization technique is *Topic Modelling*, which has already found applications in the context of TCS [23] and TCP [10]. By selecting a number of topics for a software project and using topic modelling techniques such as Latent Dirichlet Allocation (LDA) to identify which keywords belong to which topic, it is possible to represent a test case by the topic vector of its source code, i.e. the likelihood of this test case belonging to each topic. The source code of a test case has to be preprocessed in order for LDA to be able to generate meaningful topic vectors. Following the method of Thomas et al. [10], the source code of a test method is split into tokens, cleaned—Java language keywords[1], numbers, special characters and stop words are removed—, stemmed and camel case tokens are split into subtokens. This allows the model to focus on linguistic data such as code comments, identifier names and string literals [10], which is important as topic modelling has generally been applied to natural language documents and is not directly applicable to source code without adapting the process [24]. For word stemming, the *Porter Stemmer* was used as it has been utilized widely in research and consistently performs well among different stemming algorithms [25]. Tokenization, stop word removal

[1]https://docs.oracle.com/javase/specs/jls/se17/html/jls-3.html#jsl-3.9

and stemming was performed using the *NLTK* package [26]. The training of the LDA model and subsequent topic vector generation was done with the *Corpora* python package [27] using default parameters if applicable. The number of topics K is a hyperparameter and has to be chosen prior to the training process. We set K to the number of files containing test code, such that each topic corresponds to one test class (assuming each file contains one test class).

A second, more novel approach that was utilized in this paper to represent test cases as vectors is using the embeddings that are generated by LLMs. Code models such as *CodeBERT* [18] and *CodeT5+* [21] are language models that are specifically trained to understand, be able to interpret and also generate source code. Embeddings represent the model's interpretation and understanding of a given input, like the source code of a test case, which contain the contextual and semantic information that the model has gained during its training process. In this paper, four different pre-trained LLMs were chosen based on their availability and on the straightforwardness of accessing the embedding vector, namely *CodeBERT* [18], *GraphCodeBERT* [19], *UniXcoder* [20] and *CodeT5+* [21]. The embeddings were collected as specified in the documentation of the respective models[2] and all model checkpoints were retrieved via the *huggingface*[3] API.

An overview of the different possible combinations of selection strategies and distance metrics as well as which embedding data they use can be seen in Figure 1.

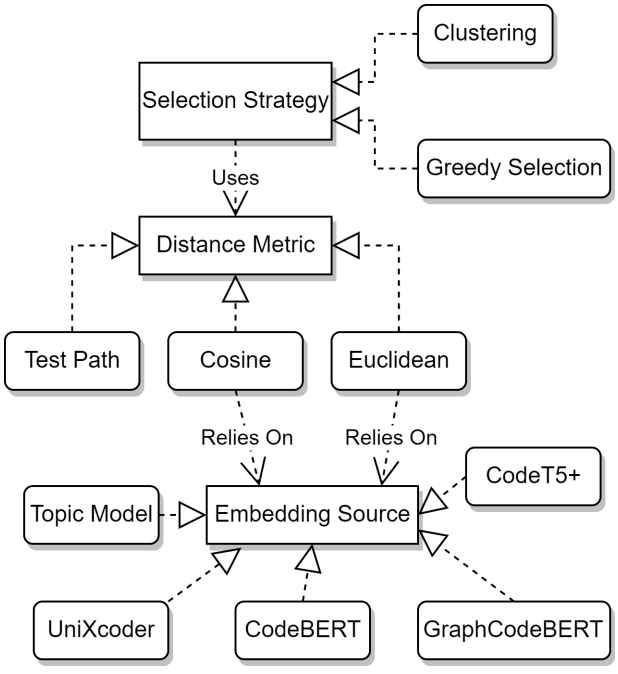

Fig. 1. The interaction between selection strategies and distance metrics

[2]https://huggingface.co/Salesforce/codet5p-110m-embedding for CodeT5+ and https://github.com/microsoft/CodeBERT for the rest

[3]https://huggingface.co/

IV. Evaluation

In this section, we present the research questions we aim to answer and detail the methodology and data we employ to do so. Furthermore we present empirical results and discuss their implications on the usability of the different selection strategies and distance metrics. Lastly, we discuss possible threats to the validity of our results.

A. Research Questions

The main application of non-coverage based TCS techniques would be to provide a cheaper and more easily maintainable alternative to coverage-based techniques. Therefore, we want to examine how much coverage our presented reduction techniques achieve and whether they are comparable to a coverage based approach. Additionally, we inspect the failure detection capabilities of our proposed approaches. We investigate the following Research Questions (RQs):

- **RQ1** How much coverage does a test suite, that is reduced by non-coverage based selection, achieve when compared to a coverage based selection?
- **RQ2** How many failures does a test suite, that is reduced by non-coverage based selection, detect when compared to a coverage based selection?
- **RQ3** How quickly do the failures occur in the test suite that is reduced by non-coverage based selection, when compared to a coverage based selection?

B. Study Design and Dataset

1) Baselines: To be able to compare and contextualize our results, we use three baselines. The first two baselines are trivial or computationally simple approaches that act as a lower boundary, meaning that we expect our TCS approaches to outperform them. The first baseline is a random selection, where tests are selected iteratively at random until the stopping criterion is met, similar to the greedy selection described in Section III-A2. The second baseline is to sort the entire suite by test execution time in ascending order and then truncate the suite at the desired point, again depending on the chosen selection criterion. For the third baseline, we use a coverage based selection that is actively used in the industry and is based on a modified version of the approach by Noemmer and Haas [9]. The modified version uses a score based on the additional coverage—coverage that a test uniquely adds to a given set of tests—as well as the execution time of a test to maximize the coverage that can be achieved in a given time frame. This baseline represents our upper bound, meaning that we expect our TCS approaches to perform the same or worse with respect to our evaluation metrics, especially for coverage where it has been shown to be very effective [28].

2) Dataset: To empirically evaluate our approaches we use one large industry and several open source projects. The open source Java projects are taken from *Defects4J*, a database containing curated projects and information about bugs including the failing tests that detected the bug and the code changes that were made to fix it [29]. It has been found to be a good dataset for evaluating failure detection

techniques, as the bugs contained are real—i.e. not seeded or mutation based—and flaky and non-deterministic failures or bugs caused by configuration issues have been selectively removed [29], [30]. We chose projects that used *JUnit4*[4], as this was required to gather the testwise coverage and other test metadata. Additionally, we focused on commits that occurred in a range of four to six weeks, as this is the interval in which coverage recording and subsequent test selection for our coverage-based baseline is done in practice.

The industry project is called *Teamscale*, which is a software quality platform written in Java and Typescript that is developed and maintained by *CQSE GmbH*[5]. The testwise coverage information required for the third baseline and to answer *RQ1* could not be recorded for the whole test suite due to the complexity of its setup, underlying the relevance of the alternative techniques we present. In particular, this included Typescript coverage generated by UI Tests and a substantial amount of system tests. To answer *RQ1*, only the tests for which testwise coverage information is available will be used (4593 tests out of 7594 in total), whereas for *RQ2* and *RQ3* only execution data containing the execution times of tests will be used. This means that for Teamscale only the random and sorting baselines will be available for these research questions. The failing tests required for *RQ2* and *RQ3* are retrieved from the project's CI pipeline. The projects that were used in our evaluation as well as important data like their size in Lines of Code (LoC) can be seen in Table I.

TABLE I
PROJECTS USED FOR EVALUATION

Project	LoC	# Tests	# Failures	*Defect4J* IDs
Teamscale	1.5M	7594	48	-
Commons Math	289.3K	3271	38	5-10, 12-24
Commons Lang	110.6K	2227	9	5-10
Mockito	95.7K	1377	14	5-9, 15-17, 21, 28-31, 34-37

3) Methodology: For each project, numerous sets of buggy commits with failing tests that occurred within four to six weeks in the CI platform are chosen. Testwise coverage on the method level is recorded once for each of these sets using a modified version[6] of the *JaCoCo* agent[7]. Test selection is performed for the most recent commit in each set using all different combinations of selection strategies and distance metrics shown in Figure 1 along with the baseline selections, where the random baseline is run 1000 times in order to account for the variance of its results.

The selection criterion used for greedy selection is the testing budget, meaning reduction is stopped after the selected tests reach a total execution time of 1%, 5% and 10% for *RQ1* and 10%, 25% and 50% for *RQ2*. Such comparatively low budgets are often enough to achieve the full coverage of a test suite, as coverage usually follows a Pareto distribution in relation to the total execution time [31] and a budget of 10% was found to result in a coverage of 100% for the coverage-based baseline in almost all cases. Empirical data suggests that the same may also apply to failure detection, as it has been found that a majority of failures can be detected using only a fraction of the whole testing budget [8]. For clustering, the duration of the selected test suite cannot be controlled, so instead we specify the reduction by the number of tests, as it has been done by some other researchers [14], [32]. We opted for a reduction of 20% and 50%. Since there is no time related stopping criterion for HAC and due to the performance of this selection strategy in regards to *RQ1*, we evaluate only the greedy selection for *RQ2*.

For *RQ3* we employ the Average Percentage of Faults Detected with Cost (APFDc) metric, which measures how quickly a test suite identifies failures by considering the position of failing tests and the execution costs in an ordered suite [6]. It has been found to be more effective when evaluating cost cognizant TCS techniques [33], [34]. APFDc is given by Equation 3, where m is the number of failures, $t_1, t_2, \ldots, t_n$ are the execution costs of the n tests and TF_i is the index of the first test in the test suite that produces failure i. The values of APFDc range from $0 - 100$, where a higher value means that the test suite finds most failures early on in the test suite execution. Since clustering gives no inherent ordering on the selected tests, we only evaluate the greedy selection for *RQ3*.

$$APFD_C = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TFi})}{\sum_{i=1}^{n} t_i} \tag{3}$$

C. Empirical Results

The results regarding the achieved coverage for all different combination of distance metric and embedding source can be seen for both selection strategies in Table II for clustering and Table III for greedy selection. Each embedding source is followed by either an *(E)* for the euclidean distance from Equation 1 or a *(C)* for the cosine dissimilarity from Equation 2. The values inside the tables denote the mean difference in retained coverage when compared to the coverage-based baseline, such that a lower value signifies a better performance result. The best performing parameter combination is highlighted in bold for each test budget.

For *RQ2* and *RQ3* the results are separated between open source projects from the *Defects4J* dataset and the industry project, due to the limitations for the latter mentioned in Section IV-B2. Table IV and Table V show the mean percentage of failures that were detected in the reduced test suite when compared to a full test suite run, for the open source and the industry project respectively.

Table VI and Table VII show the mean APFDc for the open source projects and the industry project, respectively. The values are computed for the whole test suite, i.e. no reduction is performed. Instead, a budget of 100% is specified, meaning the whole test suite will be selected and returned in the order that the greedy selection algorithm picked the tests.

[4]https://junit.org/junit4/
[5]https://teamscale.com/about-us
[6]https://github.com/cqse/teamscale-jacoco-agent
[7]https://www.jacoco.org/jacoco/trunk/doc/agent.html

TABLE II
DIFFERENCE IN RETAINED TEST COVERAGE FOR CLUSTERING COMPARED TO COVERAGE BASED SELECTION

Distance Metric	**# Tests (%)**	
	50	80
Test Path	22.05	24.62
CodeBERT (C)	22.73	25.59
CodeBERT (E)	22.66	25.62
GraphCodeBERT (C)	21.73	25.53
GraphCodeBERT (E)	21.91	25.67
UniXcoder (C)	22.86	25.62
UniXcoder (E)	23.46	25.67
CodeT5+ (C)	22.28	25.05
CodeT5+ (E)	22.33	25.20
Topic Model (C)	24.45	25.64
Topic Model (E)	24.27	25.54
Random	**7.8**	**3.82**
Time Sort	29.27	18.28

TABLE III
DIFFERENCE IN RETAINED TEST COVERAGE FOR GREEDY SELECTION COMPARED TO COVERAGE BASED SELECTION

Distance Metric	**Test Budget (%)**		
	1	5	10
Test Path	**15.38**	17.03	11.45
CodeBERT (C)	15.94	15.45	**11.10**
CodeBERT (E)	16.07	**14.98**	11.19
GraphCodeBERT (C)	15.69	15.49	11.68
GraphCodeBERT (E)	16.13	15.19	11.34
UniXcoder (C)	16.42	15.32	11.39
UniXcoder (E)	15.60	15.03	11.23
CodeT5+ (C)	15.90	15.07	11.11
CodeT5+ (E)	15.90	14.99	11.14
Topic Model (C)	15.39	15.00	11.57
Topic Model (E)	16.66	16.13	11.53
Random	40.55	44.04	42.59
Time Sort	16.56	15.16	11.61

TABLE IV
PERCENTAGE OF DETECTED FAULTS FOR GREEDY SELECTION IN OPEN SOURCE PROJECTS

Distance Metric	**Test Budget (%)**		
	10	25	50
Test Path	82.54	100	100
CodeBERT (C)	82.11	100	100
CodeBERT (E)	82.11	100	100
GraphCodeBERT (C)	82.11	100	100
GraphCodeBERT (E)	71.00	100	100
UniXcoder (C)	71.42	100	100
UniXcoder (E)	71.42	100	100
CodeT5+ (C)	71.00	100	100
CodeT5+ (E)	71.00	100	100
Topic Model (C)	82.54	100	100
Topic Model (E)	86.24	100	100
Random	21.61	31.50	50.02
Time Sort	82.54	100	100
Coverage-Based	82.26	100	100

TABLE V
PERCENTAGE OF DETECTED FAULTS FOR GREEDY SELECTION IN INDUSTRY PROJECT

Distance Metric	**Test Budget (%)**		
	10	25	50
Test Path	63.49	63.49	69.84
CodeBERT (C)	63.49	65.08	69.84
CodeBERT (E)	63.49	63.49	69.84
GraphCodeBERT (C)	63.49	63.49	71.43
GraphCodeBERT (E)	63.49	63.49	69.84
UniXcoder (C)	63.49	63.49	69.84
UniXcoder (E)	63.49	63.49	69.84
CodeT5+ (C)	63.49	63.49	68.25
CodeT5+ (E)	63.49	63.49	68.25
Topic Model (C)	63.49	63.49	68.25
Topic Model (E)	63.49	63.49	69.84
Random	9.77	24.33	49.27
Time Sort	63.49	63.49	68.25

TABLE VI
APFDC FOR GREEDY SELECTION IN OPEN SOURCE PROJECTS

Distance Metric	**APFDc**
Test Path	95.20
CodeBERT (C)	95.24
CodeBERT (E)	94.88
GraphCodeBERT (C)	94.82
GraphCodeBERT (E)	94.77
UniXcoder (C)	94.63
UniXcoder (E)	94.55
CodeT5+ (C)	94.69
CodeT5+ (E)	94.66
Topic Model (C)	95.04
Topic Model (E)	94.81
Random	50.19
Time Sort	94.60
Coverage-Based	93.70

TABLE VII
APFDC FOR GREEDY SELECTION IN INDUSTRY PROJECT

Distance Metric	**APFDc**
Test Path	72.72
CodeBERT (C)	72.01
CodeBERT (E)	72.47
GraphCodeBERT (C)	72.40
GraphCodeBERT (E)	72.52
UniXcoder (C)	72.63
UniXcoder (E)	72.81
CodeT5+ (C)	72.29
CodeT5+ (E)	72.52
Topic Model (C)	72.96
Topic Model (E)	73.45
Random	49.57
Time Sort	72.88

D. Discussion

1) Answering RQ1: Table II clearly shows that the HAC selection strategy performs very poorly, regardless of which of the distance metrics or embedding sources is used. The retained coverage is between $22 - 25\%$ less than for the coverage-based baseline and is outperformed even by a random selection. For the 50% reduction, HAC outperforms the time-sorted baseline, but when 80% of the test cases are retained it falls short of both baselines. Since HAC performs so poorly for two budgets in a broad range and is outperformed by the baselines, we conclude that this form of clustering is not a viable selection strategy when using any of the distance metrics we examined. Another disadvantage of HAC compared to greedy selection is that the execution time of a test is not considered during the clustering process itself, but only once the clusters have been generated and the test selection process begins.

For the greedy selection shown in Table III, our distance metrics consistently outperform both the random selection and the time-sorted baselines. The random baseline, which has been used in a lot of the literature on the topic as the only comparison to a proposed approach [2], [14], is outperformed by a large margin. However, it can be seen by the performance of the time-sorted baseline that achieving better coverage result than a random selection is easily achievable with an approach that requires very little computational effort. When comparing the time-sorted baseline to the results of our examined approaches, we see that while most of them outperform the baseline, they do so only by a small margin. Especially for higher testing budgets—10% in our case—the differences between all different approaches and the time-sorted baseline becomes increasingly small. This is because for most projects, and predominantly for the projects used in this evaluation, a high retained coverage can already be achieved with a comparably small testing budget [9]. The similarity in the performance of our examined approaches and the time-sorted baseline suggests that the execution time that is used to calculate the distance score (see Algorithm 1) is dominant when compared to the actual distance value. Due to the fact that all results in Table III are very similar to each other and there is no distance metric combination that consistently performs well, we can make no conclusions on whether one distance metric and embedding source combination is preferable to another. Even when comparing the two vector distances Euclidean (E) and Cosine (C) from Equation 1 and Equation 2, there is no clear winner. However, a difference of 15% in retained coverage when compared to a baseline that uses coverage information in the selection process to maximize the retained coverage is not a bad result. It shows that non coverage-based selection strategies can compete with a coverage-based approach, as long as they justify the slightly lower retained coverage with a much easier setup and maintenance process.

One key takeaway is that a selection strategy solely based on execution time, with the goal of executing as many tests as possible within a given time budget, is remarkably effective in terms of retained coverage. This type of selection strategy may not be applicable to most software project, as it would consistently choose the exact same tests each time and always neglect longer running integration or end-to-end tests. However, it provides a solid baseline for evaluating TCS or TCP approaches.

2) Answering RQ2: For the open source projects in Table IV we can see that a budget of 25% and 50% is enough to uncover all faults for the selected commits. The only variations between our different parameter combinations are in the 10% column. Here we can see that some embedding sources are in line with the time-sorted baseline while others, notably *UniXcoder* and *CodeT5+* fall short of this baseline by more than 10%. All approaches beat the random baseline by a huge margin, however almost all of them also fall below the time-sorted baseline, including the coverage-based selection. Only *Test Path* and *Topic Model* perform better than the coverage-based baseline, albeit only by a small and at times insignificant margin. These results reinforce the conclusions we drew for *RQ1*, namely that the different distance metrics struggle to differentiate themselves from the execution time that is used in the distance score calculation and that a random selection is easy to outperform and does not constitute a sensible baseline.

For the industry project in Table V, we notice that even a 50% testing budget is not enough to uncover all failures. This, combined with the fact that all distance metrics perform almost identical to the time-sorted baseline, suggests that test execution time is again the dominant factor. When investigating the failing tests that were used in the evaluation for the industry project, we indeed find that there is a higher concentration of longer running system tests—in the seconds range instead of the low milliseconds range typical of unit tests—when compared to the open source projects. This uncovers a fundamental flaw in score calculations that heavily rely on the execution time of a test as they pick the quicker unit tests first, causing longer running tests to be selected at the end and only being included for larger testing budgets. Maximizing retained coverage and detected failures for a limited test budget is a delicate balancing act that might require a selection strategy to differentiate between unit, integration and end-to-end tests. This approach could help achieve a more optimal distribution of each type of test within the selected test suite, such that longer running tests are not completely left out.

We conclude that while the tested distance metrics all perform much better than the random baseline, they do not offer a noticeable advantage when compared to a time-sorted baseline.

3) Answering RQ3: Examining the average APFDc values, which represent the speed at which failing tests are executed within the reordered test suite, we see that across the open-source projects, all proposed approaches outperform both the coverage-based and random baselines. However, there is no discernible improvement observed from the time-sort baseline to our distance metrics. In particular, the APFDc performance for the industry project lags significantly behind that of the

open-source projects. This discrepancy is likely due to the tendency of our presented distance metrics and the time-sorted baseline to prioritize unit tests with shorter execution times, thus placing them early in the reordered test suite. However, it appears that a significant number of faults in the industry project are caught by slower-running tests, such as integration or end-to-end tests. This conclusion is supported by the difference in APFDc values between the open-source projects and the industry project. In the latter, the APFDc values are significantly lower, indicating a higher occurrence of failures in the longer-running integration tests. While this issue persists in the open-source projects, its impact on the results appears to be less pronounced, as evidenced by Table IV, where a 25% test budget detects all failures, which is not the case in the industry project.

E. Threats to Validity

There are some commonly found inconsistencies or issues when it comes to studies performed in the software engineering and testing area that can compromise its results. A literature review performed by Kazmi et al. points out a number of them, which was used as a guideline to address possible issues with our evaluation methodology [2].

1) Internal Threats: The internal threats to the validity of our results mostly stem from randomness or non-deterministic data. For the greedy selection, the initial test that is selected could be different each time in the likely case that there is more than one test with the fastest execution time (e.g. $0ms$, as milliseconds are usually the smallest unit of measurement for test execution times). In addition, the code embeddings generated by the language models from Section III-B are non-deterministic by nature. Different starting tests and embedding vectors may influence the performance, although we found that this does not make a noticeable difference while examining the preliminary results.

The coverage and failure information may be inaccurate which could influence our results. This was mitigated by using the *JaCoCo* agent for coverage recording—a tool widely used in both the open source community and professional environments—as well as by utilizing failure information from a curated dataset such as *Defects4J* [29] (see Section IV-B3). The failure information used for the industry project could not be validated extensively, meaning it could contain flaky or non-deterministic test failures. As the collected failures are sampled from a real CI environment, span over many commits and are contained within a large and complex code base, the process of manually investigating each failure was too time-consuming.

The execution times for each test that is recorded in the coverage data may also vary from run to run, especially for tests that take a long time. Since our test projects contained mostly unit tests that have low execution times, this does not have a major impact in practice.

2) External Threats: As with any empirical evaluation, the generalizability of our results is not guaranteed since the possible combinations of programming languages, frameworks and testing methodologies is too vast to cover at once [35]. While in this research we only focused on tests written in Java, we tried to diversify our testing data by using industry as well as open source projects of varying sizes and application domains. Further empirical results for different projects and programming languages would be required to improve on this.

V. Conclusion

Coverage-based test selection techniques offer an intuitive approach for selecting tests, maximizing the part of the system that is executed during the testing phase for a given time budget. However, recording this coverage information can be expensive and complex [5], [7], leading to this technique being rolled out slower and only for select prioritized projects. An alternative that does not rely on coverage information, but instead on source code and meta data of the test would offer a cheaper and easier to maintain solution, that can be employed for lower prioritized projects or as a stand-in measure until the coverage-based selection is fully setup.

In this research we studied two selection techniques. HAC which has found many applications in TCS research [14], [15] and a greedy selection technique. Both selection strategies were paired with a variety of distance measures, namely the package path distance and vector distances based on a vector representation of the test's source code. For the embeddings, we investigated well known NLP techniques such as *Topic Modeling* as well as newer approaches such as embedding vectors from LLMs. All of these techniques were evaluated based on the coverage they were able to retain in the selected test suite, as well as their retained fault detection capabilities. We tested the techniques on a set of open source projects from *Defects4J*, a reputable dataset [29], [30], as well as a larger industry project in order to gain a general understanding of their performance. To put our results in context, we compared our proposed strategies to a random and a strictly time-based selection, as well as a coverage-based baseline.

We find that in terms of coverage, while our approaches are able to maintain a decent amount of the coverage when compared to a coverage-based selection, our proposed distance metrics do not achieve a noticeable improvement when compared to a time-based approach. When using HAC as a selection strategy, we note that there is a lack of a time budget stopping criterion, making this selection strategy less intuitive than other approaches. Additionally, we note that the performance of HAC in terms of retained coverage is much worse than that of the greedy selection and falls short even of the random baseline.

When it comes to fault detection capabilities, we conclude that selection strategies that rely heavily on execution times tend to neglect longer running integration and end-to-end tests, which has a negative impact on the number of faults these techniques are able to detect. This problem seems to be exacerbated for the industry project, where longer running integration and end-to-end tests are more common, suggesting that evaluating TCS strategies on a wide range of both open

source and industry applications is extremely important in order to obtain generalizable results [2].

Another conclusion that we draw is that a random selection baseline, as is often used in TCS and TCP literature [2], [14], does not represent a solid comparison as it can be easily outperformed by trivial selection techniques like a time-sort selection.

REFERENCES

[1] E. Juergens and D. Pagano, "Did We Test the Right Thing? Experiences with Test Gap Analysis in Practice," CQSE GmbH, Whitepaper, 2016.

[2] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Comput. Surv.*, vol. 50, no. 2, may 2017. [Online]. Available: https://doi.org/10.1145/3057269

[3] S. Amann and E. Jürgens, "Change-Driven Testing," in *The Future of Software Quality Assurance*, S. Goericke, Ed. Springer Verlag, 2019, ch. 1.

[4] K. Herzig, "Testing and continuous integration at scale: limits, costs, and expectations," in *Proceedings of the 11th International Workshop on Search-Based Software Testing*, ser. SBST '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 38. [Online]. Available: https://doi.org/10.1145/3194718.3194731

[5] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 91–100.

[6] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 222–232. [Online]. Available: https://doi.org/10.1145/3180155.3180210

[7] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 419–429.

[8] E. Juergens, D. Pagano, and A. Goeb, "Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites," CQSE GmbH, Whitepaper, 2018.

[9] R. Noemmer and R. Haas, "An evaluation of test suite minimization techniques," in *Software Quality: Quality Intelligence in Software and Systems Engineering*, D. Winkler, S. Biffl, D. Mendez, and J. Bergsmann, Eds. Cham: Springer International Publishing, 2020, pp. 51–66.

[10] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Softw. Engg.*, vol. 19, no. 1, p. 182–212, feb 2014. [Online]. Available: https://doi.org/10.1007/s10664-012-9219-7

[11] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 382–391.

[12] C. Fang, Z. Chen, K. Wu, and Z. Zhao, "Similarity-based test case prioritization using ordered sequences of program entities," *Software Quality Journal*, vol. 22, no. 2, p. 335–361, jun 2014. [Online]. Available: https://doi.org/10.1007/s11219-013-9224-0

[13] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[14] P. G. Sapna and H. Mohanty, "Clustering test cases to achieve effective test selection," in *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, ser. A2CWiC '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1858378.1858393

[15] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, G. Antoniol, and A. Corazza, "Clustering support for inadequate test suite reduction," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 95–105.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[19] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021.

[20] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[21] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint*, 2023.

[22] A. E. Ezugwu, A. M. Ikotun, O. O. Oyelade, L. Abualigah, J. O. Agushaka, C. I. Eke, and A. A. Akinyelu, "A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects," *Engineering Applications of Artificial Intelligence*, vol. 110, p. 104743, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095219762200046X

[23] K. Askling, "Application of topic models for test case selectionâ€¯: A comparison of similarity-based selection techniques," Ph.D. dissertation, Linköping University, 2019. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-159803

[24] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 522–531.

[25] J. Singh and V. Gupta, "Text stemming: Approaches, applications, and challenges," *ACM Comput. Surv.*, vol. 49, no. 3, sep 2016. [Online]. Available: https://doi.org/10.1145/2975608

[26] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[27] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[28] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 140–150. [Online]. Available: https://doi.org/10.1145/1273463.1273483

[29] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[30] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 130–140.

[31] R. Noemmer, "Conception and evaluation of test suiteminimization techniques for regressiontesting in practice," mathesis, Technical University of Munich, Oct. 2019.

[32] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 1–10.

[33] F. S. Ahmed, A. Majeed, T. A. Khan, and S. N. Bhatti, "Value-based cost-cognizant test case prioritization for regression testing," *PLOS ONE*, vol. 17, no. 5, pp. 1–26, 05 2022. [Online]. Available: https://doi.org/10.1371/journal.pone.0264972

[34] M. Mor, "Evaluate the effectiveness of test suite prioritization techniques using apfd metric," *IOSR Journal of Computer Engineering*, vol. 16, pp. 47–51, 01 2014.

[35] L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh, "The case for context-driven software engineering research: Generalizability is overrated," *IEEE Software*, vol. 34, no. 5, pp. 72–75, 2017.