Black-box Test Suite Minimization

by

Rongqi Pan

Thesis submitted to the Faculty of Engineering
In partial fulfillment of the requirements
For the Doctorate of Philosophy degree in
Digital Transformation and Innovation

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

In software testing, executing large test suites is time and resource-consuming, sometimes impossible, and such test suites typically contain many redundant test cases that (mostly) find the same faults. Hence, test suite minimization (TSM) is used to remove redundant test cases that are unlikely to detect new faults. However, most TSM techniques rely on code coverage (white-box), model-based features, or requirements specifications, which are not always (entirely) accessible by test engineers. Code coverage analysis also leads to scalability issues, especially when applied to large industrial systems. Recently, a set of novel techniques were proposed, called FAST-R, relying solely on test case code for TSM, which appeared to be much more efficient than white-box techniques. However, it achieved a comparably low fault detection capability for Java projects, thus making its application challenging in practice. This thesis presents two contributions addressing the key challenges in the TSM context, attempting to find better trade-offs between efficiency and fault detection.

First, we propose ATM (AST-based Test case Minimizer), a similarity-based, search-based TSM technique, taking a specific budget as input, that also relies exclusively on the source code of test cases but attempts to achieve higher fault detection through finer-grained similarity analysis and a dedicated search algorithm. The results show that ATM achieved significantly higher fault detection rates (0.82 on average), compared to FAST-R (0.61 on average) and random minimization (0.52 on average), when running only 50% of the test cases, within practically acceptable time ($1.1 - 4.3$ hours, on average, per project version).

To further improve the scalability of ATM, we propose LTM (**L**anguage model-based **T**est suite **M**inimization), a novel, scalable, and black-box similarity-based TSM approach based on large language models (LLMs). Experimental results show that the best configuration of LTM (UniXcoder/Cosine) outperforms ATM in three aspects: (a) achieving a slightly greater saving rate of testing time (41.72% versus 41.02%, on average); (b) attaining a significantly higher fault detection rate (0.84 versus 0.81, on average); and, most importantly, (c) minimizing test suites nearly five times faster on average, with higher gains for larger test suites and systems, thus achieving much higher scalability.

Acknowledgements

This work was supported by a Research Grant from Huawei Technologies Canada Company, Ltd., the Mitacs Accelerate Program, the Science Foundation Ireland under Grant 13/RC/2094-2, and the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC).

I am deeply grateful to Taher A. Ghaleb for his strong support throughout my research, especially during challenging times.

I would also like to thank the members of my Thesis Advisory Committee, Paula Branco and Tanya Schmah, for their valuable feedback, which has greatly enhanced my thesis.

Finally, I would like to express my sincere gratitude to my supervisor, Lionel Briand, for his invaluable support, encouragement, and guidance throughout my research. I truly appreciate the time and effort he dedicated to my development as a researcher.

Dedication

To my beloved cat, Seven, whose unwavering companionship and comforting presence provided me with endless support and joy throughout this journey. I couldn't have done this without you.

To my parents, whose encouragement and support have been the foundation of my academic pursuits.

To my best friend and wonderful colleague, Nazanin, who offered me endless valuable insights and accompanied me to the swimming pool, the gym, and countless fascinating places with delicious foods. You made my PhD journey rich and meaningful, helping me grow braver, more determined, and into a better version of myself.

To all my colleagues and friends in the Nanda Lab, thank you for being an integral part of my PhD journey.

Last but not least, I want to thank myself. Thank you for your relentless desire to grow, for staying true to your principles, for becoming stronger, and for pulling yourself through tough times. It is you who made yourself who you are today and made everything possible.

Table of Contents

List of Tables

List of Figures

Glossary of Acronyms

AST Abstract Syntax Tree

ATM AST-based Test suite Minimizer

BPE Byte-Pair-Encoding

FDR Fault Detection Rate

GA Genetic Algorithm

LM Language Models

LLM Large Language Models

LTM Language model-based Test suite Minimization

MT Minimization Time

NSGA-II Non-Dominated Sorting Genetic Algorithm II

SOTA State-Of-The-Art

TO Thesis Objective

TSM Test Suite Minimization

TSR Time Saving Rate

Chapter 1

Introduction

1.1 Motivation

Software testing is a widely used verification mechanism to detect faults in software releases. However, test suites tend to grow in size as software evolves, making the execution of all test cases time and resource-consuming [69], if not infeasible. Such test suites are prone to similar and redundant test cases that are unlikely to detect different faults and, if not removed, are repeatedly executed many times on many versions, such as in continuous integration contexts. This can lead to a massive waste of time and resources [69], especially for large industrial systems, thus warranting systematic and automated strategies to eliminate redundant test cases, known as test suite minimization (TSM). In contrast to test case selection [69], which selects a subset of test cases that are relevant to code changes, and test case prioritization [69], which ranks test cases based on certain characteristics, such as fault detection capability, the objective of TSM is to eliminate redundant or similar test cases that are unlikely to detect different faults [36, 69].

Though many TSM techniques exist [36], most of them rely on analyzing the test coverage of the system production code (white-box), model-based features, or requirements specifications. Despite their benefits in minimizing test suites, such information is not always (entirely) accessible or available to test engineers, making them not easy to apply in practice. Further, analyzing production code entails many scalability and practicality issues, especially when applied to large industrial systems [20,31]. One exception is the recent and novel work of Cruciani et al. [15], called FAST-R, that relies solely on the source code of test cases. Though much more efficient than white-box techniques, FAST-R achieved a relatively low fault detection capability for Java test cases. Unlike test case selection and prioritization, test suite minimization is typically performed on an occasional basis [48], that is not for every code change but rather at certain milestones, such as major releases when many new test cases are created.

Therefore, a technique that is more time-consuming than FAST-R but runs within practical time and achieves higher fault detection rates would often be a better trade-off in practice.

1.2 Thesis Objectives

This research aims to develop a black-box TSM technique that offers a better trade-off between effectiveness and efficiency than the state-of-the-art (SOTA), i.e., FAST-R. More specifically, we aim at the following two main thesis objectives (TO):

TO_1 (Improve the effectiveness of TSM): Improve the fault detection capability of TSM, when compared to the SOTA, while ensuring it runs within a practical time.

TO_2 (Improve the efficiency of TSM): Further improve the efficiency, i.e., minimization time, of TSM, while maintaining its fault detection capability.

1.3 Contributions

The research contributions of this thesis include:

1. A black-box, AST-similarity- and search-based TSM technique, called ATM, that offers a better trade-off between effectiveness and efficiency than existing work (FAST-R), in many practical contexts. This includes (a) a finer-grained technique that considers test cases to be Java test methods, pre-processes them, and transforms their code into ASTs; (b) a tree-based similarity measure that merges two complementary similarity measures that have not been used for test case similarity, thus capturing more information about test case commonalities.

2. A large-scale TSM experiment on 16 projects with 661 versions comparing several configurations of ATM and baseline techniques, taking approximately three months of calendar time and 23 years of computation time on a cluster of $1,304$ nodes with $80,912$ available CPU cores.

3. A novel black-box TSM approach (LTM) that relies, for the first time, on Large Language Models (LLMs) and two distance functions based on the generated embeddings. We investigate and conduct a comprehensive comparison among five recent language models with various model architectures, parameter sizes, inputs, and tasks used for pre-training.

4. An optimized search process of LTM by utilizing a more efficient data structure for fitness calculation, which in turn reduced the search time by 190 folds.

5. A thorough comparative analysis of efficiency and effectiveness, as well as the achieved saving in testing time, of black-box TSM approaches (LTM and ATM) based on a large-scale test suite minimization experiments involving 17 Java projects with 835 versions, thus yielding valuable insights into the relative performance of alternatives. Overall, the experiments took around three months in calendar time corresponding to 6 years of computation on a cluster with $83,216$ available CPU cores.

6. A thorough analysis of the minimization time of LTM showing that it is much more scalable than the SOTA approach (ATM) by running five times faster on average—with even higher gains for larger systems and test suites typically encountered in practice—while achieving significantly higher fault detection rates, as a result of using LLMs for embeddings.

Contributions 1-2 address TO_1, whereas contributions 3-6 address TO_2. While the author of this thesis is the main author and contributor to all of the above contributions, they also benefited from Taher A Ghaleb (postdoctoral fellow) in terms of coding, experimental design, and writing. Furthermore, all the contributions above were discussed, supervised, guided, and thoroughly reviewed by the supervisor of this thesis, Prof. Lionel Briand.

1.4 Publications

In addition to this thesis, the research results have been communicated to the public via the following publications in top software engineering venues (best conference and best journal):

1. **Rongqi Pan**, Taher. A. Ghaleb and Lionel C. Briand, "ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1700-1711, doi: 10.1109/ICSE48619.2023.00146.
2. **Rongqi Pan**, Taher. A. Ghaleb and Lionel C. Briand, "LTM: Scalable and Black-box Similarity-based Test Suite Minimization based on Language Models," in IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2024.3469582.

The detailed contributions of each publication are listed in Section 1.3, where contributions 1-2 correspond to publication 1; contributions 3-6 correspond to publication 2. We have provided a detailed statement of contributions of each co-author of the publications in Section 1.3.

1.5 Thesis Structure

The rest of this thesis is structured as follows:

- chapter 2 addresses TO_1 by providing:
  - a background on the existing TSM techniques (Section 2.1),
  - a proposed black-box TSM technique (Section 2.2),
  - a thorough empirical evaluation of the proposed technique and discussions of its practical implications (Section 2.3),

  - a discussion of threats to validity (Section 2.4),
  - a review of the related work and comparison with our proposed technique (Section 2.5),
  - a conclusion of the work (Section 2.6).

- chapter 3 addresses TO_2 by providing:
  - a background on existing black-box TSM techniques (Section 3.1),
  - a review of the related work and comparison with our proposed technique (Section 3.2),
  - a proposed TSM technique that addresses the scalability issue (Section 3.3),
  - a thorough empirical evaluation of the proposed technique and discussions of its practical implications (Section 3.4),
  - a discussion of threats to validity (Section 3.5),
  - a conclusion of our work and suggestions for future work (Section 3.6).

Chapter 2

Improving the effectiveness of test suite minimization

This chapter addresses TO_1, i.e., Improving the effectiveness, i.e., fault detection capability, of TSM technique, while making it run in a practical time. The content of this chapter has been published in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* [50].

2.1 Overview

As discussed in chapter 1, a TSM technique that achieves a better trade-off between effectiveness and efficiency, compared to FAST-R, will be a better option in practice. To achieve this goal, we propose ATM (**A**ST-based **T**est suite **M**inimizer), a TSM technique based on tree-based test code similarity and evolutionary search. ATM is black-box as it relies exclusively on test code, thus requiring no access to the production code of the system under test. ATM achieves significantly higher fault detection rates than FAST-R and runs within practical time through a finer-grained analysis of test cases and search-based optimization. ATM pre-processes and normalizes test case code, transforms it into Abstract Syntax Trees (AST), and then compares test case ASTs using four tree-based similarity measures: top-down, bottom-up, combined (merging the first two), and tree edit distance. Finally, ATM employs single-objective search, specifically Genetic Algorithm (GA) and its multi-objective counterpart, Non-Dominated Sorting Genetic Algorithm II (NSGA-II), to minimize test suite using the above similarity measures as fitness, individually or combined.

We evaluated ATM compared to baseline techniques: FAST-R and random minimization (a standard baseline). In contrast to Cruciani et al. [15], our evaluation was performed on a large set of Java test cases, extracted from 16 Java projects with many versions, each of which with a single real fault associated with one or more test case failures. Moreover, while FAST-R was evaluated at the level of Java test classes, our evaluation is finer-grained as it focuses on Java test methods, where each method is considered a test case. This was motivated by the fact that a fault may be detected by only a subset of test methods rather

than a whole test class. Therefore, performing minimization at the method level enables the removal of unnecessary test cases in a more precise manner [25, 63], which has been shown to achieve better results than those at the class level [16, 44, 71]. We used the Fault Detection Rate (*FDR*) and total minimization time (*MT*) as metrics to respectively evaluate the effectiveness and efficiency of ATM using three minimization budgets, ranging from 25% to 75%, covering most of the practical budget range as test engineers usually want to preserve significant test suite fault detection power. We compared the results of all alternative ATM configurations among themselves and, by also accounting for the time of similarity calculations, we identified the best one and compared it to baseline techniques. Specifically, we addressed the following research questions.

- *RQ1: How does ATM perform under different configurations in terms of test suite minimization?*

  For a 50% minimization budget, ATM achieved high fault detection rates (0.82 on average) and ran within practically acceptable time ($1.1-4.3$ hours on average across configurations), with combined similarity using GA being the best configuration when considering both effectiveness (0.80 *FDR*) and efficiency (1.2 hours). Such results were consistent for other budgets (25% and 75%).

- *RQ2: How does ATM compare to state-of-the-art black-box test suite minimization techniques?*

  The best configuration of ATM outperformed other techniques in terms of effectiveness, by achieving significantly higher *FDR* results than FAST-R (+0.19 on average) and random minimization (+0.28 on average), while running within practically acceptable (though longer) time (1.2 hours on average).

Contributions. Overall, this work makes the following contributions.

- A black-box, AST-similarity- and search-based TSM technique, called ATM, that offers a better trade-off between effectiveness and efficiency than existing work, in many practical contexts. This includes (a) a finer-grained technique that considers test cases to be Java test methods, pre-processes them, and transforms their code into ASTs; (b) a tree-based similarity measure that merges two complementary similarity measures that have not been used for test case similarity, thus capturing more information about test case commonalities.

- A large-scale TSM experiment on 16 projects with 661 versions comparing several configurations of ATM and baseline techniques, taking approximately three months of calendar time and 23 years of computation time on a cluster of 1,304 nodes with 80,912 available CPU cores. This is the largest TSM experiment to date in the research literature.

Chapter Structure. The rest of this chapter is organized as follows. Section 2.2 presents our proposed technique for TSM. Section 2.3 presents the experiment design, reports experimental results, and discusses their practical implications. Section 2.4 discusses the validity threats to our results. Section 2.5 reviews and contrasts our technique with related work. Section 2.6 concludes the work.

2.2 ATM: Black-box Test Suite Minimization

This section describes our black-box technique, called ATM, for TSM relying on tree-based test code similarity and evolutionary search. Figure 2.1 gives an overview of the main steps of ATM. We first describe how we pre-process the source code of test cases (Section 2.2.1) and transform them into ASTs (Section 2.2.2). Then, we describe the algorithms we employed for measuring the similarity between these ASTs (Section 2.2.3). Finally, we describe the search-based algorithms we used to minimize test suites using similarity measures as fitness (Section 2.2.4).

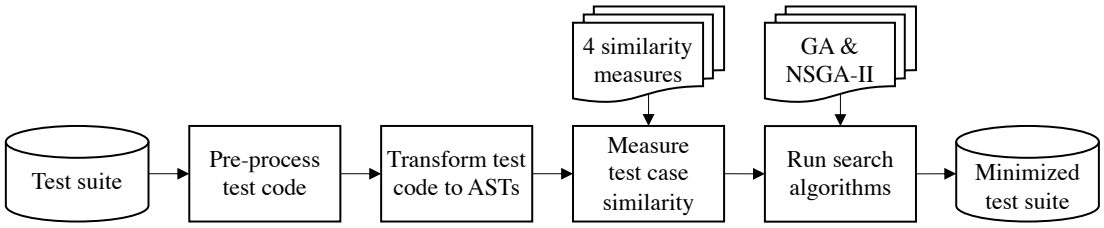


Figure 2.1: An overview of the main steps of ATM to perform TSM

2.2.1 Test Case Code Pre-processing

The source code of test cases, which is in our context Java test methods, may contain information that is irrelevant to the testing rationale, such as comments and variable names, and code statements that do not exercise the system under test, such as logging statements and test oracles (assertions). Given that we aim to compare test cases with respect to how they exercise the system, the information above is not only irrelevant but could introduce noise in our analysis. Therefore, we pre-process the source code of test cases as follows.

- We removed test case names from method declarations, since they are typically different among test cases.
- We removed Javadoc, single- and multi-line comments, since they are simply used to document test case code.
- We removed logging or printing statements, since they are simply used to record test case execution.
- We removed test oracles, i.e., assertions, similar to Silva et al. [59], since they do not exercise the system under test but rather focus on verifying the test case outcome for a given input. This includes all *JUnit* assertion methods[1], including their parameters.

[1] `https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html`

- Similar or even identical test cases can use different identifiers. Therefore, we normalized variable identifiers, rather than removing them, and retained their data types to maintain the data flow and logic of test cases. This was done by keeping track of variable and object identifiers according to their order of appearance in the test case code and then normalizing them in the form of *id*_1, *id*_2, and so on. Listing 2.1 and Listing 2.2 present the examples of two Java test cases, TC1 and TC2, respectively. Listings 2.3 and 2.4 display the test code of TC1 and TC2 after normalizing their variable identifiers. Although `TC1` and `TC2` are similar in terms of functionality, they differ in their variable identifiers, that is, `TC1` uses `a` and `b`, while `TC2` uses `x` and `y`. To normalize them, all instances of `a` and `x` are replaced with `id_1`, and all instances of `b` and `y` are replaced with `id_2`.

Test cases before normalization

```
public void TC1(int a) {
    int b = foo(a);
    if (b > 0){
        b = a * 2;
    }
}
```

Listing 2.1: Test Case 1

```
public void TC2(int x) {
    int y = foo(x);

    y = x * 2;

}
```

Listing 2.2: Test Case 2

Test cases after normalization

```
public void TC1(int id_1) {
    int id_2 = foo(id_1);
    if (id_2 > 0){
        id_2 = id_1 * 2;
    }
}
```

Listing 2.3: Test Case 1

```
public void TC2(int id_1) {
    int id_2 = foo(id_1);

    id_2 = id_1 * 2;

}
```

Listing 2.4: Test Case 2

2.2.2 Transforming Test Case Code into AST

Processing test case code as natural language using text-based or token-based techniques does not capture its syntactical information [70], thus making similarity measurement less accurate. To address this issue, we used Abstract Syntax Trees (AST) to preserve the syntactic structure of test case code [49]. To do this, we used an AST parser, provided by the Eclipse JDT library[2], to statically traverse any given test case code and transform it into a corresponding AST. Comparing the ASTs of test cases helps identify differences between them more precisely, such as differences in method calls, their number of parameters, or parameter values. AST is composed of labeled nodes, where the order of AST nodes is

[2] `https://www.eclipse.org/jdt`

important as it represents the structure of test case code. Therefore, ASTs are considered *labeled*, *ordered* trees. Figure 2.2 presents the AST representation of TC1. The AST representation captures all the syntactical information of the test case, including control structures (e.g., `if` statement), expressions (e.g., *id_2* `> 0` and *id_1* `* 2`), data types (e.g., `void` and `int`), method calls (e.g., `foo(`*id_1*`)`), identifiers (e.g., *id_2* and *id_1*), and values (e.g., `0` and `2`).

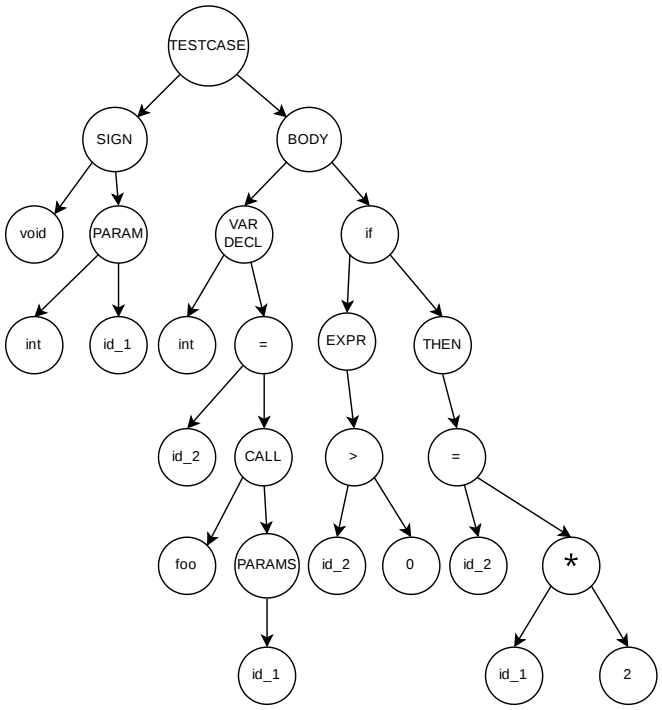


Figure 2.2: The AST representation of the test case TC1

2.2.3 Similarity Measures

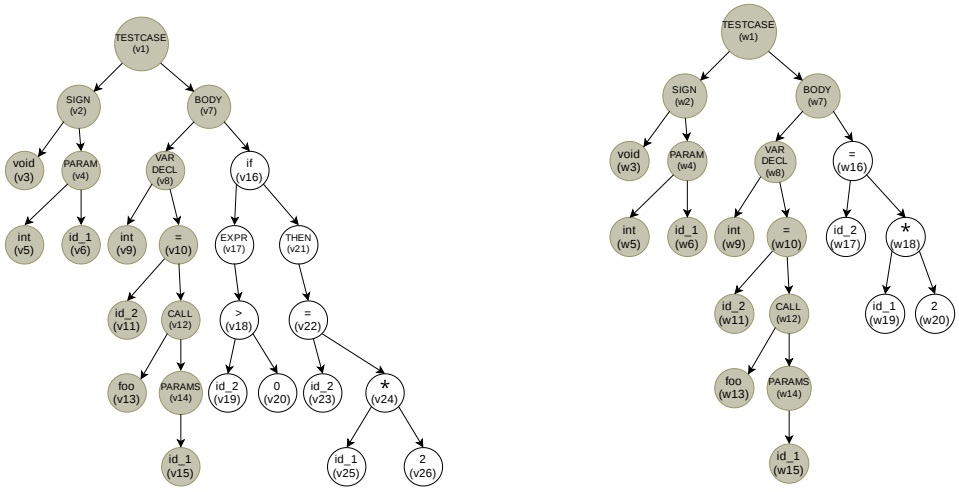
We used different algorithms to perform tree-based similarity measurements of test cases. Depending on the way trees are traversed, algorithms may capture different information from each other. Given the variety of coding conventions and practices according to which test cases are developed, a single similarity measure for TSM might perform inconsistently across projects. Therefore, we employed four similarity measures, namely *top-down* similarity (based on the top-down maximum ordered common subtree isomorphism algorithm), *bottom-up* similarity (based on the bottom-up maximum ordered common subtree isomorphism algorithm), a *combined* measure (merging the first two), and *tree edit distance* (based on the standard tree edit distance algorithm).

Both top-down and bottom-up similarity measures focus on identifying the longest common branch of the code of test cases, but in two distinct ways. Specifically, top-down

similarity is structure-oriented, thus starting the analysis at the high-level structure of test case code, e.g., iterative or conditional blocks, followed by lower-level details in a step-by-step manner. Hence, structural differences in the code of test cases can significantly impact top-down similarity. Bottom-up similarity, on the other hand, is detail-oriented, in which low-level details of the test case code, e.g., parameters to method calls, are analyzed first, thus making it less impacted by structural differences. Given that bottom-up complements top-down, we further considered merging the information obtained by both of them into a new, combined similarity measure to capture both structural and detailed aspects of test cases. Different from the above the similarity measures, tree edit distance focuses on scattered code differences between test cases rather than a common code branch and aims to capture all code changes that can make one test case similar to another. We describe below each of the similarity measures and point to the book of Valiente [62] for more details.

Top-down similarity measure Figure 2.3 shows the ASTs of `TC1` and `TC2`, denoted as AST_1 and AST_2, respectively, and highlights the top-down maximum common subtree between the two ASTs. The top-down similarity measure is based on the top-down maximum common subtree isomorphism algorithm [62]. Tree isomorphism determines whether the nodes of one tree have a bijective correspondence with the nodes of another tree. Given that AST represents structured source code, in which the order and type of code statements is important, we considered labeled, ordered tree isomorphism. A top-down maximum common subtree isomorphism between two labeled, ordered trees is obtained by traversing them simultaneously using *preorder traversal*. Preorder traversal ensures that the parent of each node is included in the subtree, since they are visited first, thus resulting in a top-down subtree. Top-down similarity does not capture similar subtrees with different parent nodes. Typical examples include the same block of code but with a different loop, such as `for` and `while`, or calls to two different methods with the same parameters.

Bottom-Up similarity measure Figure 2.4 shows the ASTs of `TC1` and `TC2`, denoted as AST_1 and AST_2, respectively, and highlights the bottom-up maximum ordered common subtree between them. The bottom-up similarity measure is based on the bottom-up maximum common subtree isomorphism algorithm [62]. A bottom-up maximum ordered common subtree isomorphism between two labeled, ordered trees is obtained by first partitioning tree nodes into equivalence classes, and then finding nodes in both trees belonging to the same equivalence class with the largest size. The size of an equivalence class is equal to the size of the bottom-up subtree rooted at a specific node. Two nodes belong to the same equivalence class if and only if the bottom-up ordered subtrees rooted at them are isomorphic [62]. A *postorder traversal* is then performed on T_1 and T_2 to partition children nodes into equivalence classes, starting with one tree to populate a dictionary of equivalence classes, followed by the second tree with the same dictionary shared. The equivalence class of a visited node is obtained from the dictionary if its label and the equivalence classes of its children nodes already exist. Once equivalence classes are assigned to all nodes of the two trees, the maximum bottom-up common subtree is obtained based on the nodes sharing the same equivalence class and having the largest bottom-up subtree rooted at them. The bottom-up similarity measure can capture information that the top-down maximum common subtree algorithm might not. For example, in contrast to top-down similarity, if



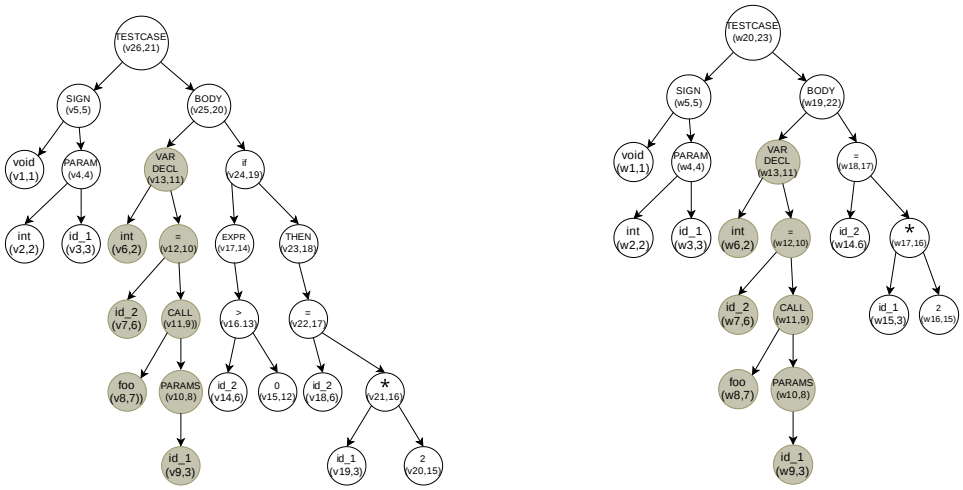
(a) AST_1 (b) AST_2

Figure 2.3: A top-down maximum common subtree of two ASTs: AST_1 and AST_2. The nodes are numbered according to the preorder traversal. The common subtree is represented by the highlighted nodes in both ASTs.

two test cases call two different methods with the same parameters, then the matching parameters of method calls are included in the bottom-up maximum common subtree. Also, if some children of the nodes in the bottom-up ordered subtrees do not match, then these nodes are not included as part of the common subtree.

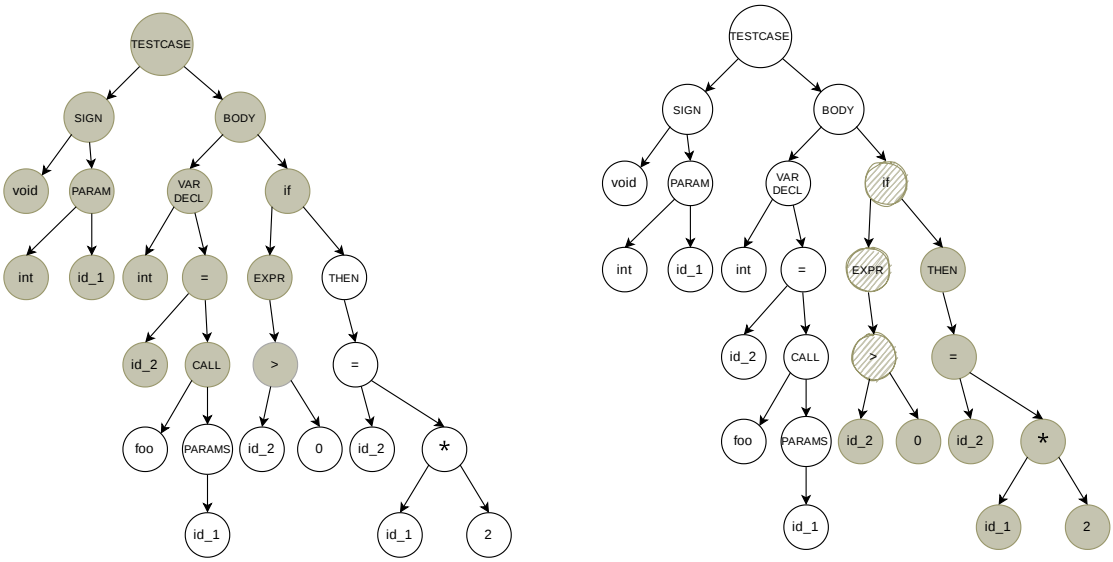
Combined similarity measure (top-down+bottom-up) Given that top-down and bottom-up common subtrees capture different and complementary information in the test case code, we also considered merging their resulting subtrees into a single, combined common subtree. Combining top-down and bottom-up maximum common subtrees was performed by taking the union of nodes in both subtrees, while eliminating overlapping nodes. For each node of one subtree, we checked whether it is present in the other subtree by considering its label and the labels of its parent, siblings, and children nodes. If there is a match, then an overlap is identified and thus not included as part of the combined common subtree. The size of the combined common subtree is equal to the sum of the size of the unique nodes of the top-down and bottom-up common subtrees, where overlapping nodes are discarded. As shown in Figure 2.5, the size of the top-down and bottom-up common subtrees is 13 and 11, respectively. There are three overlapping node, i.e., `if`, `EXPR`, `>`. Hence, the size of the combined common tree is 13 + 11 - 3 = 23.

Tree edit distance similarity measure This measure is based on the edit distance algorithm, which calculates the total number of elementary edit operations, i.e., insertion, deletion, and substitution of nodes, required to convert one tree into another tree [62], which is commonly used for tree comparison. To do this, a sequence of elementary edit operations is applied to one tree until the other tree is obtained. Tree edit distance is not



(a) AST_1 (b) AST_2

Figure 2.4: A bottom-up maximum common subtree of two ASTs: AST_1 and AST_2. Nodes are numbered according to the postorder traversal as well as their equivalent classes. The common subtree of AST_1 and AST_2 is represented by the highlighted nodes in both ASTs.



(a) Top-down Common Subtree (b) Bottom-up Common Subtree

Figure 2.5: The combined common subtree that merges the top-down common subtree and bottom-up common subtree of two ASTs.

expected to be efficient when compared to previous similarity measures, especially for large ASTs, but is nevertheless an option. An simple example is shown in Figure 2.6, where the AST_1 is transformed into AST_2 by applying a sequence of deletion and insertion operations.

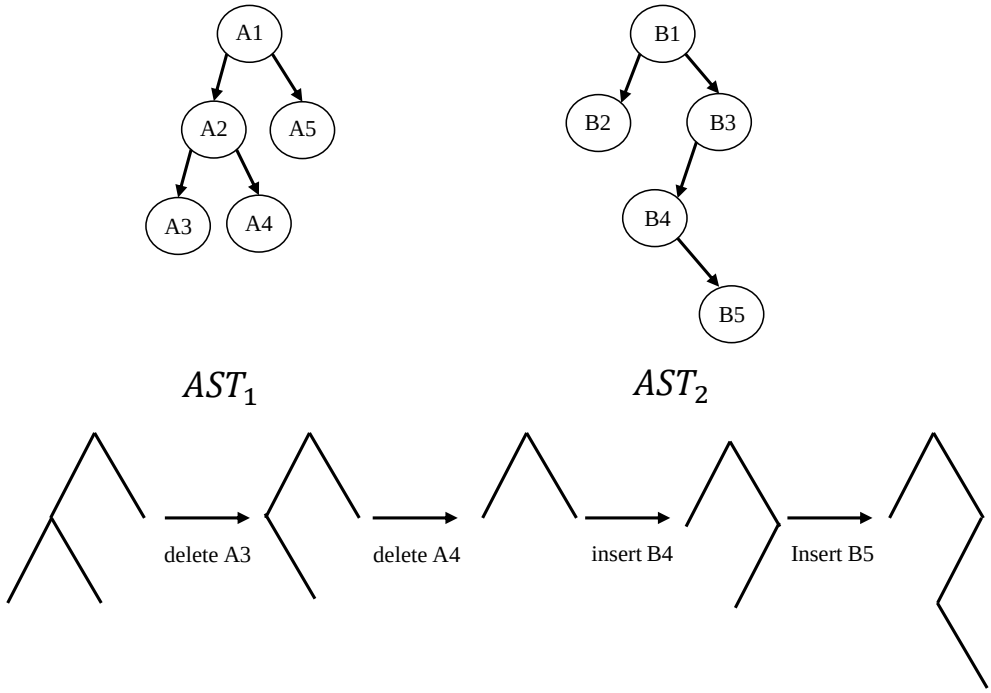


Figure 2.6: An example of a conversion between AST_1 and AST_2.

Similarity score calculation Similarity scores for the top-down, bottom-up, and combined similarity measures were calculated the same way, but differently from tree edit distance, as described below.

- For top-down, bottom-up, and combined similarity measures, after identifying their maximum ordered common subtrees, a similarity score was calculated as follows.

$$Sim_{MaxCommonSubtree}(T_1, T_2) = \frac{2 \times |V_m|}{|V_1| + |V_2|} \tag{2.1}$$

  where T_1 and T_2 are two trees with a total number of $|V_1|$ and $|V_2|$ nodes, respectively. V_m is the number of nodes included in the maximum ordered common subtree.

- For tree edit distance, the similarity score was calculated as follows.

$$Sim_{TreeEditDistance}(T_1, T_2) = \frac{|V_1| + |V_2| - d}{|V_1| + |V_2|} \tag{2.2}$$

  where T_1 and T_2 are two trees with a total number of $|V_1|$ and $|V_2|$ nodes, respectively, and d is the number of tree edit operations.

Similarity measurement implementation We used an open-source library, called simpack[3] for implementing the algorithms for top-down and bottom-up maximum ordered

[3] `https://files.ifi.uzh.ch/ddis/oldweb/ddis/research/simpack`

common subtree isomorphism and tree edit distance. However, for the bottom-up maximum ordered common subtree isomorphism, we extended the library to support labeled trees [62] (available in our replication package [1]). We did so by assigning unique integers to node labels, which are then used to assign equivalence classes for identifying the maximum bottom-up common subtree of two labeled, ordered trees.

2.2.4 Search-based Test Suite Minimization

Considering that TSM is an NP-hard problem, we employed meta-heuristic search algorithms to help find near-optimal, feasible solutions for this problem. Meta-heuristic techniques enable us to efficiently explore the search space of minimized test suites. Given that ATM relies on test case similarity, a search algorithm can help identify and eliminate most redundant test cases, thus producing a more diverse subset of the test suite for a given budget. We employed two search algorithms: Genetic Algorithm (GA) and its multi-objective counterpart, namely Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [17].

Genetic Algorithm (GA) This is the most widely used search algorithm in search-based software testing, inspired by evolutionary theory. Applying GA requires properly defining (a) individuals or chromosomes referring to the possible solutions, which are minimized test suites in our context, and (b) a fitness or objective function to assess the quality of each individual with respect to an optimization problem, which is TSM in our context, where test cases with higher similarity values are eliminated. To tailor GA to our problem, we re-formulated the optimization problem as a fixed-size subset selection problem. To do this, we represented a minimized test suite as a binary vector whose length equals the total number of test cases before minimization, where 1 means a test case is included and 0 means otherwise. We selected the following three genetic operators: (1) *selection*, for which we used a binary tournament selection to select the test subset with the lower fitness among two subsets; (2) *crossover*, for which we used a customized crossover operator [8] that keeps only test cases belonging to both parent test subsets, then proceeds with the remaining test cases, and repeats this process until a certain number of test cases is reached; and (3) *mutation*, for which we used a permutation operator to randomly select a segment of the individual and rearrange the test cases, which are represented by 1s and 0s, within the segment through the permutation operation [8]. This operator preserves the number of selected test cases in the individual, thus ensuring the fixed-size offspring. The parameters used for our GA are consistent with what is recommended in published guidelines[4]. Specifically, we used a population size of 100, a mutation rate of 0.01, and a crossover rate of 0.90. The GA evolution process is repeated until a termination criterion is satisfied, which is in our case when the fitness value improves by less than 0.0025 with a minimum of 30 generations. Once the termination criterion is reached, minimization stops and the final minimized test suite is expected to contain diverse test cases.

Non-Dominated Sorting Genetic Algorithm II (NSGA-II) This is a multi-objective alternative of GA that is based on the *Pareto* dominance theory [17]. We used NSGA-II to

[4] `https://www.obitko.com/tutorials/genetic-algorithms/recommendations.php`

consider two similarity measures at once in our test subset selection, where each objective consists in minimizing a distinct similarity measure. Individual ***A*** *Pareto* dominates individual ***B*** if individual ***A*** is at least as good as individual ***B*** in all objectives, and superior to individual ***B*** in at least one objective [42]. To do this, all individuals are sorted into several *Pareto* non-dominated fronts and a *Pareto* front rank is assigned to each individual. We used a binary tournament selection operator, which selects test cases based on (a) the *Pareto* non-domination front ranks of individuals, and (b) the crowding distance measuring the density of individuals around a particular individual. When two individuals have the same *Pareto* front rank, the one with the highest crowding distance is selected. We used the same operators and termination criterion as for GA. We considered two alternative pairs of similarity measures as fitness: (1) top-down & bottom-up, to determine whether considering them as independent fitness functions leads to better results than combining their subtrees into a single similarity score (combined), and (2) combined & tree edit distance to assess whether the latter complements the longest common subtrees in identifying test case similarity.

2.3 Validation

This section reports on the experiments we conducted to evaluate the effectiveness and efficiency of ATM. As mentioned in the introduction, the focus here is on Java projects given the unsatisfactory results obtained by previous work, as discussed in related work. We discuss below the research questions we addressed, the experimental design and dataset, and the results achieved with their practical implications.

2.3.1 Research Questions

RQ1: How does ATM perform under different configurations in terms of test suite minimization?

TSM aims to remove redundant test cases with a minimal fault detection loss within practically reasonable time. However, its performance can be influenced by the similarity measure used to compare test cases to each other. In this RQ, we assess the performance, in terms of both effectiveness and efficiency, of ATM under various configurations, each with a different combination of similarity measures, either individually (using GA) or together (using NSGA-II). We evaluated the alternative ATM configurations on each Java project using three minimization budgets (25%, 50%, and 75%) widely covering the minimization range. In addition, we analyzed the trade-off between effectiveness and efficiency of the alternative ATM configurations. Specifically, we addressed the following sub-RQs.

- ***RQ1.1: How effectively can ATM minimize test suites?***

  A similarity measure determines what information about test cases should be used to compare them. However, such information varies widely from one similarity measure to another, which can in turn affect the fault detection capability of a TSM technique.

In this RQ, we assess the effectiveness of ATM in terms of fault detection capability for the tree minimization budgets.

- ***RQ1.2: How efficiently can ATM minimize test suites?***

  TSM should be practically scalable to projects with large test suites. However, given that similarity algorithms traverse test case ASTs differently, their execution time can vary, thus affecting the overall time required to minimize test suites. In this RQ, we assess the efficiency of ATM in terms of preparation time (taken for transforming the code of test cases into ASTs and calculating similarity scores) and search time (taken for running search algorithms).

RQ2: How does ATM compare to state-of-the-art black-box test suite minimization techniques?

Selecting test cases arbitrarily can indeed reduce test suites, but is not a viable option as it does not take test case similarity and fault detection capability into consideration. To address this issue, many TSM techniques [36], both white-box and black-box, were proposed to remove redundant test cases that are likely to detect the same faults. However, white-box techniques rely on production code, which is not always (entirely) accessible or available to test engineers and entails scalability and practicality issues in many contexts. Further, as described in Section 2.5, despite the high efficiency of some existing black-box techniques, their fault detection loss was observed to be relatively high for Java test cases, thus making them ineffective in practice. In this RQ, we assess the performance of ATM compared to two baseline techniques: (a) random test suite minimization, a standard baseline, and (b) FAST-R, a set of of novel black-box techniques, whose efficiency was shown to be much higher than white-box techniques while achieving a comparable low fault detection capability for Java test cases. Further, given that both efficiency and effectiveness are important factors in selecting minimization techniques in practice, we discuss the trade-offs between fault detection rate and minimization time.

2.3.2 Experimental Design and Dataset

We performed a series of experiments to evaluate the performance of ATM and assess the most effective and efficient configurations across various combinations of similarity measures and search algorithms, and compared the best configuration to baseline techniques. Each technique was applied to each project version, independently, and was thus run $6,610$ times (661 projects' versions $\times$ 10 runs). We considered all projects' versions to increase the number of instances in our experimental evaluation. However, in practice, minimization is expected to be applied only when required, i.e., many new test cases are created. All experiments were performed on a cluster of $1,304$ nodes with $80,912$ available CPU cores, each with a $2x$ AMD Rome 7532 with 2.40 GHz CPU, 256M cache L3, 249GB RAM, running CentOS 7. Overall, our experiments took approximately three months of calendar time and 23 years of computation time. Moreover, to mitigate randomness in the obtained results, we ran each experiment 10 times, each with a different random number generator

seed, ranging from 1 to 10, to enable the reproduction of our results. The reported results are summarized for all runs using descriptive statistics.

Minimization budgets

For experimental purposes, the minimization budgets were set at 25%, 50%, and 75% of the test suites. Our choice of minimization budgets was constrained by both the very large computation time of our experiments and the fact that test engineers, according to our discussion with industry partners, usually want to preserve significant fault detection power, knowing that prioritization and selection techniques can ultimately be used to further reduce testing time. We focus our analysis and discussion on the results obtained for the 50% minimization budget as such a percentage is sufficiently large to challenge the minimization algorithms and to show the practical significance of ATM compared to other techniques. Results for the other two minimization budgets were consistent in terms of observations and conclusions, and can be found in our replication package [1].

Baseline techniques

Random minimization. We used random minimization of test suites as a standard baseline in our evaluation. It is considered the simplest technique and is commonly used as a baseline of comparison for more sophisticated search algorithms [3]. It randomly generates subsets of test cases for any given minimization budget. Similar to other techniques, we ran random minimization 10 times, using fixed seeds ranging from 1 to 10 to enable the reproduction of results. We then calculated the average fault detection rate (*FDR*) across projects' versions.

FAST-R. We compared ATM to FAST-R, a set of four novel black-box TSM techniques proposed by Cruciani et al. [15], namely FAST++, FAST-CS, FAST-pw, and FAST-all. All FAST-R alternatives rely solely on the code of test cases. FAST++ and FAST-CS convert test case code into vectors using term frequency [61], and based on these vectors, test cases are then clustered using k-means++ [6] (FAST++) and constructed coresets [7] (FAST-CS). FAST-pw and FAST-all, however, use minhashing and locality-sensitive hashing [54] to identify diverse test cases based on Jaccard distance and random sampling, respectively.

While FAST-R performed very efficiently, it achieved relatively low median *FDR* results when evaluated on Java projects, ranging from 0% to 22% for minimization budgets from 1% to 30%. This motivated us to compare the performance of ATM to FAST-R on test cases collected from a larger set of Java projects with many versions. Though ATM uses finer-grained information from test case code, which is likely to require longer time to execute, we aim to investigate whether it can achieve significantly higher *FDR* results within practical minimization time. We compared ATM to FAST-R using the three minimization budgets indicated above and the same evaluation procedures and metrics. We relied on the publicly available implementation of FAST-R[5] provided by its authors. Though

[5] `https://github.com/ICSE19-FAST-R/FAST-R`

FAST-R was originally evaluated on Java test classes, it could easily be adapted to Java test methods, since it takes as input (a) test code, regardless of its granularity, and (b) a mapping of faults and test cases, the former being easily mapped to test methods rather than test classes.

Note that we did not compare with white-box techniques, since analyzing code coverage for all test cases of all project versions would be computationally challenging and is unnecessary given that the *FDR* results of FAST-R have been shown to be comparable to white-box techniques.

Dataset

We evaluated ATM compared to the baseline techniques on 16 Java projects collected from a public dataset, called DEFECTS4J[6], the same source of Java projects used to evaluate FAST-R. Only one project from DEFECTS4J was left out as it was far too large to consider for running our experiments, which already undertook months of computations. While FAST-R was evaluated on five Java and five C projects, each with a single version, it achieved relatively lower fault detection rates on Java projects compared to C projects, where Java test cases are test classes and C test cases are command lines. Therefore, in this chapter, we focused the implementation of ATM on Java projects and assessed it on a much larger dataset of Java projects and versions. Each project has multiple (faulty) versions, ranging from 4 to 112, with many test cases each, ranging from 152 to $3,916$. Each project version contains a single *real* fault associated with one or more test case failures, and was fixed by modifying the production code. We acknowledge that minimization should ideally be evaluated on project versions with multiple faults to achieve higher realism, but there exist no such public datasets with *real* faults. Further, with multiple faults per version, some faults may mask other faults and it becomes very hard to determine which test case detects which fault, thus making experiments rather complicated. Though our dataset is much larger in terms of systems and test cases than any previous TSM experiment, we fully realize that industrial systems can be much larger. They would, however, be unusable in our experiment as they would take far too much time. The scalability issue is further discussed in Section 2.3.4. It is therefore easy to determine whether a test suite detects a particular fault in a given version: at least one test case fails. We used this dataset to evaluate ATM and baseline techniques.

Different from the FAST-R's original study, we performed our evaluation on all faulty project versions. In addition, the evaluation of FAST-R on Java projects was performed at the class level where test cases are Java test classes, each of which group test methods exercising similar functionalities. This makes it impossible to identify redundant test methods within the same test class. Further, removing a whole test class can be misleading as one fault may be detected by only a subset of test methods in the test class. Thus TSM at the method level helps remove unnecessary test cases in a more precise manner [25, 63], which has been shown to achieve better results than those at the class level [16, 44, 71].

[6] `https://github.com/rjust/Defects4J`

Therefore, our evaluation of ATM is finer-grained as each test case is considered to be a Java test method.

We extracted the source code of test methods for each version of the projects and mapped each fault to its corresponding failing test method(s). Then, we transformed the test case code into ASTs, which were saved in XML format. After that, we calculated the similarity scores for each pair of test cases using the similarity measures described in Section 2.2.3. To reduce the time required for similarity calculation in each version during our computationally intensive experiments, we calculated the similarity scores for all test cases in the first version of each project. Then, for subsequent versions, we calculated similarity scores for only test cases in changed or newly added test files, whereas similarity scores for unchanged ones were obtained from previous versions.

Evaluation metrics

Fault Detection Rate (FDR). TSM aims to remove redundant test cases for a given budget while maintaining high *FDR*. Therefore, we used *FDR* to assess the effectiveness of ATM. Given that each project version contains a single fault, the corresponding fault detection rate per version can either be 1 (fault detected) or 0 (fault undetected). Therefore, we calculate the fault detection rate for each project by considering all its versions, as follows:

$$FDR = \frac{\sum_{i=1}^{m} f_i}{m} \tag{2.3}$$

where m denotes the total number of versions of the project. For a project version i, f_i is equal to 1 if at least one failing test case is included in the minimized test suite, indicating the detection of the fault of that version, or 0 otherwise.

For example, in the *Chart* project, which has a total of 26 faulty versions, if the minimized test suites detected the faults in 21 versions but did not in 5 versions, *FDR* would be calculated as 21/26, resulting in an *FDR* of 0.81. Note that we ran our experiments for each version a total of 10 times to mitigate randomness. Hence, the final *FDR* for each project is obtained by taking the average *FDR* results across the 10 runs.

Fisher's exact test. We used Fisher's exact test [55], a non-parametric statistical hypothesis test, to assess how significant is the difference in proportions of detected faults between the alternative ATM configurations.

Table 2.1 presents the contingency table for the Fisher's exact test. For each ATM configuration, there are 6610 samples (661 project versions × 10 runs per version). The first row indicates the number of samples that detected the fault after minimization, while the second row shows the number of samples that did not detect the fault. A p-value is then computed using Fisher's exact test based on the data in this contingency table to assess the significance of differences in proportions across configurations.

Odds ratio. We used the odds ratio [4] as an effect size measure of the magnitude of improvement of one ATM configuration over another. An odds ratio of 1 indicates no

	Config.1	Config.2	Total
Fault Detected	5395	5232	10627
Fault Not Detected	1215	1378	2593
Total	6610	6610	13220

Table 2.1: An example of a contingency table for Fisher's exact test. The columns represent two configurations of ATM, each with 6,610 samples (661 project versions × 10 runs). The first row shows the number of faults detected, and the second row shows the number of faults not detected.

difference between two techniques, whereas an odds ratio of > 1 indicates a higher chance for one technique to perform better than the other.

Total minimization time (MT). Minimization time has significant practical implications for large systems and test suites. Therefore, we assessed the efficiency of ATM by computing (1) the *preparation time*, taken to transform the code of test cases into ASTs and calculate similarity between all pairs of test cases, and (2) the *search time*, taken to run search algorithms. Note that, when reporting MT results, we did not consider the savings in MT that can be achieved by only calculating similarity between new pairs of test cases but rather accounted for all test cases in each project version independently from other versions. This, of course, makes ATM look worse in terms of MT.

For each ATM configuration, we computed the average MT for each project version.

We also computed the MTs taken by the baseline techniques.

Similarity measures as fitness

Our fitness functions add up similarity scores for all test case pairs, normalize the summation by the number of test case pairs in a $[0-1]$ range, and thus quantify how similar overall test cases are in a given test suite regardless of its size. However, we could consider a test case to be redundant if it is highly similar to at least one other test case, thus making it unnecessary to consider the other test cases. Therefore, taking the pair with the maximum similarity score for each test case as a fitness value could better distinguish highly redundant test cases. Moreover, in a similar vein, we could consider that only very high similarity scores truly matter in terms of test cases being redundant and that simply summing up similarity scores among pairs is not the best measurement for minimization. Therefore, we could give more weight to test cases with higher similarity scores by squaring or exponentiating such scores. For example, though 0.9 and 0.8 scores have the same difference (0.1) as 0.4 and 0.3 scores, their relative difference significantly increases after taking their squares (0.17 vs. 0.07) or exponentials (0.23 vs. 0.14). As a result, removing a test case with a higher similarity score leads to a greater reduction in fitness. Finally, for the reasons invoked above, we could also take into account, for each test case, only the pair with the maximum squared or exponentiated similarity score, thus focusing on highly redundant test cases. In our experiments, we considered all the above alternatives

Table 2.2: Descriptive statistics of *FDR* and *MT* (in minutes) of LTM across project versions for the 50% minimization budget. The highest *FDR* and shortest *MT* are highlighted in bold

Statistic \ Technique	GA								NSGA-II			
	Top-Down		Bottom-Up		Combined		Tree Edit Distance		Top-Down & Bottom-Up		Combined & Tree Edit Distance	
	FDR	*MT*	*FDR*	*MT*	*FDR*	*MT*	*FDR*	*MT*	*FDR*	*MT*	*FDR*	*MT*
Min	0.53	0.42	0.56	**0.28**	0.58	0.46	**0.70**	0.66	0.60	1.11	**0.70**	1.38
25% Quantile	0.75	1.95	0.68	**1.33**	0.75	2.18	0.76	3.79	0.74	5.67	**0.78**	7.12
Mean	0.78	70.87	0.74	**67.05**	0.80	72.75	0.81	82.23	0.78	235.41	**0.82**	258.44
Median	0.79	12.53	0.72	**7.76**	0.79	13.86	**0.82**	16.61	0.79	37.01	**0.82**	37.15
75% Quantile	0.84	57.88	0.81	**40.27**	**0.88**	63.01	**0.88**	77.77	0.82	187.74	**0.88**	208.38
Max	0.93	642.31	0.90	706.31	**0.97**	641.42	0.93	**491.52**	**0.97**	2295.20	0.92	2384.56

for fitness and results showed that using the maximum of squared similarity scores, shown below, achieves the highest *FDR*.

$$Fitness = \frac{\sum_{i,t_i \in M_n} Max_{i,j,t_i,t_j \in M_n, i \neq j} \; Sim(t_i, t_j)^2}{n} \tag{2.4}$$

where M_n is a minimized test suite of n test cases, and $Sim(t_i, t_j)$ is the similarity score for each pair of test cases t_i and t_j.

2.3.3 Results

We focus our discussion on the results achieved using the maximum of squared similarity scores as fitness for the 50% minimization budget (results for other budgets lead to identical conclusions and are available in our replication package [1]).

RQ1 results

Table 2.2 reports the *FDR* and *MT* (in minutes) for ATM using GA and NSGA-II using the four similarity measures, individually and combined, for the 50% minimization budget.

RQ1.1 results. Table 2.2 shows that all ATM configurations achieved high *FDR* results (mean $\geq$ 0.74 and median $\geq$ 0.72 across projects, for a 50% minimization budget). The highest average *FDR* was achieved by NSGA-II with combined & tree edit distance (mean and median = 0.82), and ranging from 0.70 to 0.92 across projects. The difference in *FDR* between projects needs further investigation, as it may be attributed to variability in numbers of faults, test suite sizes, or test coding conventions, tentative explanations that remain to be confirmed. Overall, detecting over 80% of faults when executing 50% of test cases is encouraging and, as discussed below, significantly outperforms baseline techniques.

Moreover, ATM using GA achieved a higher *FDR* with top-down (+0.04 on average) than bottom-up across projects' versions, except for the *JacksonXml* project where the former achieved 0.25 lower average *FDR* than the latter. One possible explanation is that, unlike other projects, failing test cases in *JacksonXml* have higher maximum top-down similarity scores than other test cases, which indicates high redundancy among them, thus leading to the removal of some of them. Our results also show that ATM using GA with

combined similarity yielded an even higher *FDR* (mean = 0.80, median = 0.79) than with top-down, only 0.01 and 0.02 lower than GA with tree edit distance and NSGA-II with combined & tree edit distance, respectively. This suggests that taking the union of the top-down and bottom-up common subtrees helped capture additional, relevant information about test case commonalities. Further, ATM using GA with combined similarity achieved a higher *FDR* than NSGA-II with top-down & bottom-up (0.78), thus suggesting that a simpler minimization algorithm (GA) with a single similarity measure (combined similarity) outperforms a more sophisticated and expensive algorithm (NSGA-II) relying on two similarity measures.

Fisher's exact test results revealed no significant *FDR* differences between ATM using GA with combined similarity and both GA with tree edit distance and NSGA-II with combined & tree edit distance ($p-value > 0.05$). More details about the results of Fisher's exact test can be found in our replication package [1]. In other words, even though GA with combined similarity yielded a slightly lower ($0.01 - 0.02$ less) *FDR* on average, there is no evidence that this difference is statistically significant as results vary across projects' versions. Compared to GA with tree edit distance, GA with combined similarity achieved a higher *FDR* for four projects (*Collections*, *Csv*, *JacksonDatabind*, and *Time*) and the same *FDR* for four projects (*Cli*, *Compress*, *Jsoup*, and *JxPath*). Overall, our results suggest that, when accounting only for *FDR*, ATM alternatives using either GA or NSGA-II, with tree edit distance and/or combined similarity as fitness, are roughly equivalent for all minimization budgets.

RQ1.2 results. The *MT* results in Table 2.2 combine both preparation time and search time for each project version (see detailed results in our replication package [1]). We observe that the average *MT* for the whole ATM process using GA, with all configurations, ranges from 1.1 to 1.4 hours on average per project version (with a much lower median of $7.8-16.6$ minutes due to two relatively larger projects). Given the application context of TSM, where redundant test cases are removed on an occasional basis when many new test cases are created for major releases, such *MT* are acceptable in practice, as further discussed below. However, the *MT* of ATM using NSGA-II was nearly three times longer than that of GA (mean = $3.9 - 4.3$ hours and median = 37 minutes). This result suggests that ATM using NSGA-II with two similarity measures fares much worse in terms of *MT*, while offering no significant improvement in terms of *FDR*.

Preparation Time. We found that the time ATM took for transforming test case code into ASTs was negligible (< 1% of total time), whereas the time for calculating similarity scores was much longer. Specifically, the average preparation time for calculating top-down, bottom-up, and combined similarity scores for each pair of test cases across project versions was $3.42e^{-06}s$, $4.93e^{-05}s$, and $1.75e^{-04}s$, respectively. In sharp contrast, though achieving the highest *FDR* when using GA, we found that tree edit distance, as expected, took at least an order of magnitude longer to calculate than other similarity measures (an average of 35 minutes per each project version, compared to 3 minutes for combined similarity). Therefore, on larger projects with much larger test suites and ASTs, such as those commonly found in the industry, the absolute computation time difference between tree edit distance and other similarity measures is expected to be large and probably

Table 2.3: Descriptive statistics of *FDR* and *MT* (in seconds) for the 50% minimization budget, across projects' versions, of ATM using GA with combined similarity compared to FAST-R and random minimization. The highest *FDR* and shortest *MT* are highlighted in bold

Statistic \ Technique	ATM GA/Combined		FAST++		FAST-CS		FAST-pw		FAST-all		Random minimization	
	FDR	*MT*	*FDR*	*MT*	*FDR*	*MT*	*FDR*	*MT*	*FDR*	*MT*	*FDR*	*MT*
Min	**0.58**	27.58	0.51	0.06	0.48	0.06	0.25	0.45	0.38	0.42	0.17	**0.0012**
25% Quantile	**0.75**	130.83	0.57	0.10	0.57	0.08	0.38	0.97	0.54	0.90	0.44	**0.0013**
Mean	**0.80**	4,364.76	0.61	0.44	0.60	0.20	0.47	4.14	0.59	2.78	0.52	**0.0021**
Median	**0.79**	831.71	0.60	0.19	0.60	0.14	0.47	2.21	0.62	1.82	0.50	**0.0017**
75% Quantile	**0.88**	3,780.37	0.65	0.63	0.64	0.29	0.54	6.37	0.66	4.27	0.57	**0.0025**
Max	**0.97**	38,485.15	0.72	2.17	0.71	0.63	0.72	17.11	0.70	9.10	1.00	**0.0056**

crucial in terms of applicability. Hence, in terms of preparation time, ATM with combined similarity is more scalable and a better alternative than tree edit distance in practice.

Search Time. We found that the search time ATM took for finding the optimal minimized test suites using NSGA-II with both top-down & bottom-up and with combined & tree edit distance (mean = 3.7−3.9 hours and median = 31−37 minutes) was about three times longer than that of GA with combined similarity. In addition, we found that, though took much longer to calculate similarity, tree edit distance took less search time, since it converged faster, than combined similarity using GA. However, when considering the total *MT* for the whole process, ATM ran faster when using GA with combined similarity for all projects, except for ***Time***. As a result, ATM using GA with combined similarity is more scalable than with tree edit distance and far more efficient than using NSGA-II, while achieving comparable *FDR* results, thus making it the best configuration. This can be particularly important on large projects and test suites.

RQ1 summary. ATM achieved high *FDR* results (0.82 on average) and ran within practically acceptable time (1.1−4.3 hours on average) when running 50% of test cases, with combined similarity using GA being the best configuration when considering both effectiveness (0.80 *FDR* on average) and efficiency (1.2 hours on average). Results were consistent for other minimization budgets (25% and 75%).

RQ2 results

Table 2.3 compares, in terms of *FDR* and *MT* for the 50% minimization budget, the best ATM configuration (GA with combined similarity) to the baseline techniques: FAST-R (FAST++, FAST-CS, FAST-pw, and FAST-all) and random minimization.

***FDR* results.** We observe that ATM using GA with combined similarity systematically outperformed random minimization with a significantly higher *FDR* (+0.28 on average). In addition, all FAST-R alternatives, except FAST-pw, outperformed random minimization, with FAST++ being the best FAST-R alternative (0.61 on average), followed by FAST-CS (0.60 on average). However, compared to ATM (GA with combined similarity), all FAST-R alternatives achieved significantly lower *FDR* results across projects, with

an average *FDR* difference of −0.19 between FAST++ and ATM, thus making ATM much more effective in practice.

***MT* results.** We observe that all FAST-R alternatives ran much faster than ATM, in terms of both preparation and search time, with an average *MT* of $0.20 - 4.14$ seconds across projects' versions (FAST-CS was the fastest technique, with 0.20 seconds). However, given its considerably low *FDR*, i.e., missing about 40% of faults when executing 50% of test cases, FAST-R is not a practically viable option in many contexts.[7] TSM is typically performed on an occasional basis [48], usually at certain milestones, such as new major releases when many new test cases are created. Therefore, given that ATM achieves much higher *FDR* and runs within practically acceptable (though longer) time, it is the most advantageous choice in many practical contexts.

RQ2 summary. ATM outperformed baseline techniques by achieving significantly higher *FDR* results than FAST-R (+0.19 on average) and random minimization (+0.28 on average), while running within practically acceptable (though longer) time (1.2 hours on average) given the application context.

2.3.4 Discussion

Effective test suite minimization with easily accessible information. Our results showed that ATM performs significantly better than baseline techniques, thus enabling test engineers to run test suites for a desired budget while being more likely to maintain an acceptable *FDR*. This is done without requiring production code analysis, a significant practical advantage in many contexts. While ATM achieved high *FDR* result using similarity measures, both individually and combined, there is still room for improvement in terms of *FDR*, which could be achieved by considering additional similarity measures, thus capturing various and complementary aspects relevant to test case commonalities. Also, similarity measurement in ATM considered a variety of potentially relevant information in test case code. For example, similarity between method calls considers their names, number of parameters, and parameter values. Disregarding some of these details could result in a higher similarity and better results. The importance of these details may also differ from one project to another, which suggests that test engineers might need to tailor the definition of similarity to their needs and context.

Effective versus efficient test suite minimization. Our results showed that GA with combined similarity is the best configuration for ATM, given its effectiveness and efficiency compared to other configurations. Specifically, it achieved a high *FDR* that is comparable to GA with tree edit distance and NSGA-II alternatives, while taking much less time to calculate. However, despite the considerably longer time taken to calculate tree edit distance compared to combined similarity, it took relatively less time to search

[7] As a side note, though out of the scope of this work, FAST-R achieved lower average *FDR* results (up to 0.56, achieved by FAST++) and took slightly longer ($0.23 - 5.75$ seconds) when evaluated at the class level (Java test classes), as in the original FAST-R study.

for the optimal test suite as it converges faster on all projects. Still, when accounting for the total minimization time, ATM using GA with combined similarity ran faster than tree edit distance on the majority of the projects, thus making it the best configuration. Moreover, though very efficient, random minimization and FAST-R alternatives performed significantly worse than ATM in terms *FDR*. Given that TSM is typically performed on an occasional basis, such as major releases with many new test cases, even if ATM takes much longer than FAST-R due to its finer-grained similarity measures and search algorithms, it still runs in practical time (1.2 hours on average) and achieves much higher *FDR*, thus making it a better choice in many practical contexts.

Scalability. On the largest project in our dataset, ***Time***, which has nearly $4k$ test cases, ATM took more than 10 hours, on average, per version, largely due to the search producing an optimal minimized test suite. Though our TSM experiment is the largest to date, industrial systems can be much larger than the ones in our dataset. Overall, we observed that the search time of the search algorithm increases quadratically with the number of test cases, regardless of the similarity measure. As a result, there is obviously a limit, in terms of system and test suite sizes, to any minimization technique. Therefore, future research should devise ways of pushing the scalability boundary of ATM further while preserving high *FDR* results. For instance, similarity and search fitness computations can be easily parallelized to significantly improve scalability. Indeed, all similarity computations of test case pairs and fitness values of minimized sets in a population are independent and can be run on different cores [9, 35]. Other search algorithms [42] should also be investigated to identify better trade-offs between efficiency and effectiveness.

Application of minimization in practice. In practice, when there is a major release with many new test cases created and a testing budget is set, similarity values of new test case pairs are calculated. Then, search is performed to find a subset of the test suite that minimizes the similarity between test cases within the given test budget. This minimized test suite is later used for regression testing in the subsequent code versions.

2.4 Threats to validity

Construct Validity Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. Test cases may contain information that is irrelevant to the testing rationale, which could introduce noise when measuring similarity between test cases. To mitigate this threat, ATM pre-processed test cases and transformed their code into ASTs.

Internal Validity Internal threats to validity are concerned with the ability to draw conclusions from our experimental results. Compared to FAST-R, ATM was evaluated on finer-grained data: test cases were considered to be Java test methods instead of test classes. However, given that FAST-R was originally evaluated based on Java test classes, its performance on test methods could be inconsistent in our experiments. To mitigate this threat, we evaluated all techniques, ATM, FAST-R, and random minimization, on the same finer-grained data. We observe that FAST-R's performance was consistent with

what was originally reported [15], in terms of both effectiveness and efficiency, and even worse when evaluated at the class level. Moreover, to mitigate randomness in the obtained results, we ran each experiment 10 times, with fixed seeds to enable the reproduction of our results. Finally, we removed all assertions, including their parameters as we assume they do not include any method calls that exercise the system under test (with side effects) as their main goal is to verify the test case outcome. Calling methods with side effects inside assertions is not considered a good testing practice. Test cases may become similar in our context when assertions are removed, meaning they exercise the same behavior of the system, resulting in excluding one of them. However, assertions might still contain relevant information regarding test case similarity and retaining them might increase FDR. Future research should further investigate this point.

External Validity External threats are concerned with the ability to generalize our results. Our evaluation was performed on a large dataset extracted from 16 Java projects collected from Defects4J, including 661 faulty versions. However, unlike FAST-R, we did not evaluate ATM on C projects, though it can a priori be applied to test cases written in other languages provided the availability of tools for transforming test case code into ASTs. Future research should assess ATM's performance on test cases written in other programming languages to devise more general conclusions.

2.5 Related Work

TSM is an NP-hard problem, hence there are many techniques that have been proposed [36, 69] to find near-optimal, feasible solutions to address it. Such techniques can be classified into three categories: greedy heuristics-based, clustering-based, and search-based.

Greedy heuristics-based test suite minimization. Miranda et al. [46] used greedy heuristics [12] to perform TSM that iteratively selects test cases based on their code coverage until all the target program entities, e.g., statements, for a given testing input are covered. This technique achieved fault detection rates ranging from 0.52 to 0.69 for running 2.6% to 11.3% of test cases. Noemmer et al. [48] also used greedy heuristics to perform TSM using statement coverage. Mutation testing was used to evaluate this technique, which obtained a loss in mutation scores ranging from 3.5% to 21% for running 7% to 50% of test cases. However, these techniques require to analyze production code, i.e., white-box, and do not enable targeting specific minimization budgets, thus having scalability and applicability issues in practice.

Clustering-based test suite minimization. Cruciani et al. [15] recently proposed a novel technique, called FAST-R, to perform TSM by relying solely on test case code (black-box), which is converted into vectors using a term frequency model [61]. Based on these vectors, clustering was used to partition test cases into clusters, where the centroids of the clusters were selected as the minimized set of test cases. It was evaluated on five Java and five C projects, each with a single version. Though it was more efficient than white-box techniques, it achieved relatively low median fault detection rates on Java projects, ranging from 0.18 to 0.22 for minimization budgets ranging from 1% to 30%. In

addition, the experiment considered Java test classes to be test cases, where test methods exercising similar functionalities are grouped together, thus making it impossible to identify redundant test methods within the same test class. Similarly, Coviello et al. [14] and Viggiato et al. [65] proposed clustering-based TSM techniques relying on code coverage or test cases written in natural language, thus making them not easy to apply in practice as such information are not always accessible by test engineers.

Search-based test suite minimization. Hemmati et al. [30] used a search algorithm that relies on test case similarity calculated using model-based features to perform TSM. This technique was evaluated on two relatively small industrial systems and achieved average fault detection rates ranging from 0.60 to 1.00 for running 4% to 14% of test cases. Similarly, Zhang et al. [72] and Wang et al. [67] proposed model-based TSM techniques relying on multi-objective search algorithms, such as NSGA-II, MOCell [47], SPEA2 [73]. However, the information required by the above techniques is not always available to test engineers.

Summary. In contrast to the above techniques, except for FAST-R, ATM relies exclusively on test case code and can help test engineers target any pre-set minimization budget, thus making it more applicable in practice. Compared to FAST-R, ATM achieved significantly higher (+0.19) fault detection rates within practically acceptable (though significantly longer) *MT*. Moreover, ATM was evaluated on a larger, finer-grained dataset of 16 Java projects with 661 faulty versions, thus making it by far the largest experiment to date for TSM.

2.6 Conclusion

In this chapter, we proposed ATM, a black-box test suite minimization (TSM) technique based on the Abstract Syntax Tree (AST) similarity of test code and evolutionary search algorithms. We investigated four tree-based similarity measures, namely top-down, bottom-up, combined (merging the first two), and tree edit distance. We employed Genetic Algorithms (GA) and its multi-objective counterpart (NSGA-II), to perform TSM using the above similarity measures to define alternative fitness functions. We evaluated various configurations of ATM on a large dataset of 16 Java projects with 661 (faulty) versions collected from Defects4J. We used the Fault Detection Rate (*FDR*) and Total Minimization Time (*MT*) evaluation metrics to respectively assess the effectiveness and efficiency of ATM, using three practical minimization budgets ranging from 25% to 75%. We identified the best ATM configuration and compared it to FAST-R, a recently proposed set of black-box TSM techniques, and random minimization, a standard baseline. For a minimization budget of 50%, we observed that all ATM configurations achieved significantly higher *FDR* results (0.82 on average) compared to FAST-R (0.61 on average) and random minimization (0.52 on average). In addition, all ATM configurations ran within practically acceptable time ($1.1-4.3$ hours on average), with combined similarity using GA being the best configuration when considering both effectiveness and efficiency (1.2 hours on average). Results were consistent for other budgets (25% and 75%).

Chapter 3

Improving the efficiency of test suite minimization

This chapter addresses TO_2, i.e., improving the efficiency, i.e., minimization time, of TSM technique. The contents of this chapter have been published in the Journal of *Transactions on Software Engineering (TSE)* [51].

3.1 Overview

As discussed in chapter 2, we proposed ATM, which is a TSM technique that achieves better trade-off between effectiveness (i.e., *FDR*) and efficiency (i.e., *MT*) than the SOTA approach, FAST-R. However, ATM still presents scalability issues for very large software systems, since its total minimization time (the sum of preparation time and search time) increases rapidly with test suite size, represented by the number of test cases per project version. Our analysis of the total minimization time of ATM reveals that similarity measures play a major role in limiting its scalability, due to the fact that (1) computing tree-based similarity is expensive, taking up to 41.20% of the total minimization time, and (2) similarity measures impact the search convergence and speed. This motivated us to investigate similarity measures that are both more efficient to calculate and more informative to guide the search.

Language models pre-trained on programming languages convert test code into vector-based embeddings. This enables vector-based similarity measurement, which facilitates implementation optimizations and results in much higher computational efficiency than tree-based similarity measurement that relies on traversing AST trees. Moreover, these language models are pre-trained on large source code corpora with various code understanding and generation tasks, thus generating embeddings that capture informative syntactic and contextual details from test code. This suggests that such embeddings with vector-based similarity might be potentially more informative than tree-based similarity in guiding evolutionary search. Therefore, in this chapter, we propose LTM (**L**anguage model-based **T**est suite **M**inimization), a scalable, black-box similarity-based TSM approach that is based on

pre-trained Large Language Models (LLMs) and vector-based similarity measures, making it the first application of LLMs in the context of TSM, to the best of our knowledge. LTM uses the source code of test cases (Java test methods), without requiring any preprocessing, as input to five alternative pre-trained language models, namely *CodeBERT* [23], *GraphCodeBERT* [28], *UniXcoder* [27], StarEncoder [39], and CodeLlama [57], in order to extract test method embeddings. Considering that test suite diversity was reported to have a positive correlation with fault detection [2,29,30], LTM employs two similarity measures: *Cosine similarity* and *Euclidean distance*, to calculate the similarity between the extracted embeddings. Using the calculated similarity values, LTM employs a Genetic Algorithm (GA) to minimize test suites, which searches for the most optimal subset of a test suite, for a given testing budget. We evaluated LTM with the same dataset used to evaluate ATM (DEFECTS4J), followed a similar experimental design, and used the same evaluation metrics of effectiveness and efficiency: Fault Detection Rate (*FDR*) and Minimization Time (*MT*), respectively. In addition, considering the potential variation in test case execution times, assessing minimized test suites based on their number of test cases only might not always be accurate [36]. Therefore, we extended our evaluation of minimized test suites using an additional metric: Time Saving Rate (*TSR*). Moreover, we optimized GA by utilizing a more efficient data structure to accelerate fitness calculation and enhance memory usage, which led to a 190-fold reduction in minimization time, without requiring any additional computation resources. Then, we identified the best configuration of LTM by considering both effectiveness and efficiency, among all its alternatives, and compared it to the two best ATM configurations. Finally, we expanded our experiments by running LTM on a much larger project that ATM could not handle, to further assess the scalability of our approach.

Specifically, we address the following research questions.

- *RQ1: How does LTM perform for test suite minimization under different configurations?* LTM achieves high *FDR* results (an overall average *FDR* of 0.79 across configurations) for a 50% minimization budget (i.e., the percentage of test cases retained in the minimized test suite). The best configuration of LTM is UniXcoder using Cosine similarity when considering both effectiveness (0.84 *FDR* on average) and efficiency (0.82 min on average), which also achieves a greater time saving rate (an average *TSR* of 41.72%). For the large project, ***Closure***, UniXcoder using Cosine Similarity takes only 17.80 min in terms of *MT* and achieves an *FDR* of 0.79, while saving 52.55% of testing time.

- *RQ2: How does LTM compare to ATM?* The best configuration of LTM outperforms ATM by achieving significantly better *FDR* (0.84 versus 0.81, on average), and more importantly, running much faster than ATM (0.82 min versus 4.06 min, on average), in terms of both preparation time (up to two orders of magnitude faster) and search time (up to one order of magnitude faster). The latter is particularly important on typically large industrial systems and test suites where such differences practically matter.

To summarize, the contributions of this work are as follows:

- We propose a novel black-box TSM approach (LTM) that relies, for the first time, on LLMs and two distance functions based on the generated embeddings. We investigate and conduct a comprehensive comparison among five recent language models with various model architectures, parameter sizes, inputs, and tasks used for pre-training.
- We optimize the search process of LTM by utilizing a more efficient data structure for fitness calculation, which in turn reduced the search time by 190 folds.
- We conduct a thorough comparative analysis of the efficiency and effectiveness, as well as the achieved saving in testing time, of black-box TSM approaches based on a large-scale test suite minimization experiments involving 17 Java projects with 835 versions, thus yielding valuable insights into the relative performance of alternatives. Overall, the experiments took around three months in calendar time corresponding to 6 years of computation on a cluster with 83,216 available CPU cores.
- We analyze the minimization time of LTM and show that it is much more scalable than the state-of-the-art (SOTA) approaches by running five times faster on average—with even higher gains for larger systems and test suites typically encountered in practice—while achieving significantly higher fault detection rates, as a result of using LLMs for embeddings.

Chapter Structure The rest of this chapter is organized as follows. Section 3.2 reviews related work and motivates our research for this chapter. Section 3.3 describes our test suite minimization approach. Section 3.4 validates our approach, reports the experimental design and results, and discusses the implications in practice. Section 3.5 discusses the validity threats to our results. Section 3.6 draws conclusions and suggests future work.

3.2 Related Work

Test suite minimization (TSM) aims at improving the efficiency of software testing by removing redundant test cases, thus reducing testing time and resources while maintaining the effectiveness of the test suite (i.e., fault detection capability) [36]. There are various approaches that have been proposed to address TSM, including (a) greedy heuristics-based approaches [46, 48], which select test cases iteratively based on code coverage information, (b) clustering-based approaches [14, 40], which group test cases based on the similarity of their coverage information, and (c) search-based approaches [30, 72], which employ evolutionary search to find an optimal subset of the test suite using model-based features. Most of these approaches utilize code coverage information (white-box) [69] (i.e., requires access to the system production code) or model-based features, which are not always available to test engineers [5, 50]. Moreover, collecting code coverage information can result in up to 30% time overhead [15, 31], making it not scalable for large test suites and systems.

Viggiato et al. [65] investigate the similarity of test cases that are written in natural language. Though their objective is to identify redundancy among test cases, as opposed

to TSM within a budget, their similarity measures can priori be further used for this purpose. They convert test cases into vector-based representations using five text embedding techniques (i.e., Word2Vec [45], BERT [18], Sentence-BERT [56], Universal Sentence Encoder [10], and TF-IDF (Term Frequency–Inverse Document Frequency) [33]). They then identify similar test cases utilizing clustering algorithms to group similar test steps based on two different similarity metrics (Word Mover's Distance (WMD) [38] and Cosine Similarity). The results show that their approach achieves an F-Score of 83.47% for identifying similar test cases. In summary, different from our work, their focus is on identifying redundant test cases written in natural language, as opposed to code, and they do not aim to optimize a test suite within a budget.

Philip et al. [53] proposed a black-box approach, called FastLane, which relies on test and commit logs containing historical information, to predict test case outcomes (i.e., *pass* or *fail*) using logistic regression to skip their execution. Using such information, FastLane reduces testing time and resources by running only a subset of test cases instead of the whole test suite. Results showed that FastLane reached 99.99% in terms of test outcome accuracy and saved up to 18.04% of testing time. However, it requires historical information about multiple runs of test cases, thus making FastLane inapplicable for new or recent test cases. Moreover, FastLane cannot be easily adapted to given minimization budgets (i.e., a predefined percentage of test cases to be executed). Arrieta et al. [5] relied on test case inputs and outputs for simulation-based testing and employed Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [17] to find an optimal subset of test cases as a minimized test suite. Chang et al. [11] detected redundancy in test cases written in natural language using word embedding techniques and Cosine similarity. However, such information is not accessible in many contexts, thus rendering such black-box approaches not applicable.

Cruciani et al. [15] proposed a black-box approach, called FAST-R, that relies solely on the source code of test cases. FAST-R converts test code into vectors using a term frequency model [61] and then employs a random projection technique [34] to reduce the dimensionality of vectors. Based on these vectors, clustering-based algorithms were performed with the centroids of clusters selected as a minimized test suite. Results showed that, compared to white-box approaches, FAST-R was much more efficient in terms of total minimization time while achieving comparable effectiveness in terms of fault detection capability. However, FAST-R achieved relatively lower fault detection capability for Java projects, with median fault detection rates ranging from 0.18 to 0.22 for minimization budgets ranging from 1% to 30%. With such low effectiveness, FAST-R is therefore not a viable option in many practical contexts.

To achieve a better trade-off between the effectiveness and efficiency of test suite minimization, Pan et al. [50] proposed a black-box test suite minimization approach, called ATM, which relies on test code similarity and evolutionary search. ATM preprocessed test code by removing the information that is irrelevant to the testing rationale and then converted it into Abstract Syntax Trees (ASTs). Then, four different tree-based similarity measures (i.e., top-down, bottom-up, combined, and tree edit distance) were used to calculate similarity values between these ASTs. Finally, evolutionary search (i.e., GA and NSGA-II) was employed using the similarity values as fitness to find, for a given test budget, an optimal subset of the test suite that contains diverse test cases. Results showed

that the best configuration of ATM (i.e., GA with combined similarity) achieved a better trade-off in terms of effectiveness and efficiency of test suite minimization than FAST-R by achieving a significantly higher average fault detection rate (+19%) while running within practically acceptable time (1.2 *hours* on average), thus making it a better option in many contexts compared to FAST-R.

However, ATM suffers from limitations regarding its scalability for very large projects in the dataset, such as *Time*, which consists of $30k$ lines of code in its most recent version with large test suites (nearly $4k$ test cases per version), as minimization took more than 10 hours per project version as compared to FAST-R (2.17 seconds) and random minimization (0.006 seconds). The similarity measures used by ATM had a direct impact on its scalability. First, the similarity calculation time took up to 41.20% (using tree edit distance) of the overall minimization time of ATM, which is due to the fact that converting test code into ASTs and calculating tree-based similarity based on ASTs are both time- and resource-consuming. Second, the search time, which was influenced by the employed similarity measures and the number of test cases per version, increased rapidly with the test suite size. Interestingly, though tree edit distance took much longer to calculate than other ATM similarity measures, it enabled GA search to converge faster towards the termination criterion (a fitness improvement of less than 0.0025 across generations). Although GA with combined similarity was identified as the best ATM configuration based on the average minimization time across projects (1.2 *hours*), it is important to note that it was 2.5 *hours* slower than GA with tree edit distance for the largest test suite with nearly $4k$ test cases. Therefore, when dealing with large test suites, adopting a more informative similarity measure that facilitates faster search convergence can result in a notable reduction in minimization time, regardless of the computational cost associated with the similarity calculation.

In contrast to the above TSM approaches, LTM relies on test code without requiring any preprocessing, and employs large pre-trained language models (i.e., CodeBERT, GraphCodeBERT, UniXcoder, StarEncoder, and CodeLlama) and commonly used vector-based similarity measures (i.e., Cosine similarity and Euclidean distance) [66] to generate more informative similarity values between test case pairs. The goal is to both calculate similarity more efficiently and better guide the search algorithm (GA) to converge faster to a better fault detection capability.

3.3 LTM: Language Model-based Test Suite Minimization

This section describes our approach for test suite minimization, called LTM, relying on language models to compute test code similarity and enable the use of evolutionary search for minimization. Our goal is to find a black-box solution that achieves a better trade-off, in terms of scalability and effectiveness, than the latest state-of-the-art (SOTA) approach, namely ATM [50]. Despite the higher fault detection rates achieved by ATM, it has limitations in terms of scalability, which are due to the transformation of test cases into ASTs

and the time-consuming calculation of tree-based similarity, making it difficult to apply for very large software systems. Similar to ATM, LTM is also a similarity-based approach operating under the assumption that there exists a positive correlation between test suite diversity and its fault detection capability [30]. We aim to find a similarity measure that is not only more computationally efficient but also provides better guidance for the search to converge faster to a higher fault detection capability and thus result in less search time. To achieve this, we employed pre-trained language models to generate code embeddings and used them as the basis to measure the vector-based similarity between test case pairs. The computation of vector-based similarity is much more efficient than tree-based similarity requiring tree traversal. Moreover, language models have the capability to capture patterns based on the syntax and semantics of test case code without requiring preprocessing or feature extraction, making them a more suitable alternative in our context.

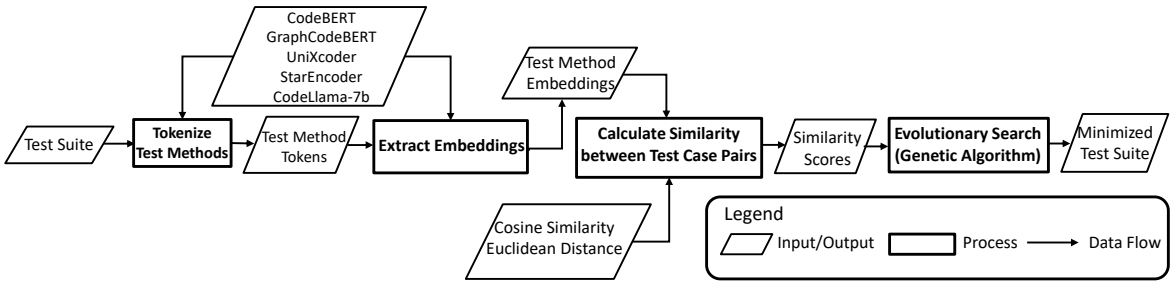


Figure 3.1: Main steps of LTM to perform test suite minimization

Figure 3.1 outlines the main steps of LTM to minimize test suites. Regarding the first two steps, we describe how we tokenized the source code of test cases (Section 3.3.1) and then extracted embeddings using language models (Section 3.3.1). Then, we describe the algorithms (Cosine Similarity and Euclidean Distance) we employed for measuring the similarity between these embeddings (Section 3.3.2). Finally, we describe the search algorithm (Genetic Algorithm) we employed to minimize test suites using the similarity measures as fitness.

3.3.1 Language Models for Test Method Representation

ATM used ASTs (represented in XML format) to preserve the syntactic structure of test case code and calculated tree-based similarity between the ASTs of test case pairs, which we observed to be time- and resource-consuming due to the need for tree traversal, thus leading to scalability issues. In addition, we found that among all tree-based similarity measures, ATM using tree edit distance achieved the highest *FDR* and required the least search time across all project versions, since it made the search algorithm converge faster towards the termination criterion, that is fitness improvement is less than 0.0025 across generations. This suggests that, regardless of the computational cost associated with its calculation, tree edit distance is a more informative similarity measure as it can be more effective in guiding the search to converge faster towards a higher fault detection rate. Therefore, we aim to develop a solution that overcomes the scalability issues of ATM

by finding similarity measures that (1) do not require test code preprocessing, (2) are computationally efficient, and (3) are more informative, offering better guidance to the search algorithm and accelerating its convergence, thus reducing search time.

In this chapter, we employ five pre-trained language models, namely CodeBERT [23], GraphCodeBERT [28], UniXcoder [27], StarEncoder [39], and CodeLlama [57]. We select these models for the following reasons:

- CodeBERT, GraphCodeBERT, and UniXcoder are open-source publicly available language models that are specifically designed for programming languages and pre-trained on a large source code corpus. These language models surpassed the performance of other SOTA models in code clone detection tasks [23, 27, 28, 41], relying on the similarity between pairs of code fragments [27], thus making them viable options for similarity-based test suite minimization.
- Similar to CodeBERT and its follow-up models, StarEncoder is a language model designed for understanding and analyzing code. Though it was not assessed for code clone detection tasks, StarEncoder is more recent and was pre-trained on a dataset encompassing more diverse (86) programming languages, enabling it to recognize test case patterns written in different coding styles.
- Unlike the above models that are pre-trained for code understanding tasks, CodeLlama is specialized for code generation and infilling tasks (i.e., filling a missing part of a code snippet). It has billions of parameters, which is significantly higher than the parameter size ($125\boldsymbol{Million}$) of the above models, and was pre-trained on a larger dataset, making it potentially able to distinguish test code more precisely.

In our context, these language models take the source code of test cases as input, without requiring any preprocessing or transformation into other formats, and generate embeddings that capture both semantic and contextual information from test case code. The output of these language models is represented as numeric vectors, which offer the opportunity to employ a variety of vector-based algorithms to measure similarities between test cases. The process starts by passing the source code of a test case (a Java test method in our context) as input to the language models, which is then converted to a list of tokens, each of which is assigned an individual numerical vector representation. The final output is a vector representation (embedding) generated for the test method, which aggregates the information from all individual vector representations, which is called a test method embedding in our context.

Tokenizing Test Methods

Language models deal with a test code fragment (method) as a sequence of tokens and split it using byte-pair-encoding (BPE) [58], which is a subword segmentation algorithm that splits words into a sequence of sub-words. This enables language models to better handle out-of-vocabulary words, such as method and variable names. For example, the tokenizer

splits the method name *testCloning* into $\dot{G}test$ (where $\dot{G}$ denotes a space), *Cl* and *oning*, since the *testCloning* word does not exist in the vocabulary and is thus separated into these three sub-words. The generated tokens are then processed as follows.

- CodeBERT, GraphCodeBERT, and StarEncoder add two special tokens, namely $[CLS]$ and $[SEP]$, to the beginning and end of each sequence of tokens, respectively. The CLS token is a special token that represents the whole input sequence, whereas the $[SEP]$ token is a separator token that denotes the end of the sequence [18]. In summary, each test method is represented as $[CLS], c_1, c_2, ..., c_m, [SEP]$, where c_i denotes the i^{th} code token and m is the total number of code tokens.
- For UniXcoder, besides the $[CLS]$ and $[SEP]$ tokens, an additional special token ($[Enc]$, $[Dec]$, or $[E2D]$) is added to the beginning of the input indicating the pre-training mode of the model as encoder-only, decoder-only, or encoder-decoder mode, respectively. These modes differ in terms of model architectures and training tasks during pre-training, thus supporting various downstream tasks. We used the encoder-only mode for producing contextualized code embeddings, as decoder-related modes are used for code generation [27]. Therefore, the final input for UniXcoder is represented as $[CLS], [Enc], [SEP], c_1, c_2, ..., c_m, [SEP]$.
- CodeLlama uses <*s*> and </*s*> tokens denoting the beginning and end of each token sequence, respectively: (<*s*>, $c_1, c_2, ..., c_m$,</*s*>).

During pre-training, each token is then mapped to a 768-dimensional vector representation that contains the semantic and contextual information of this token for all language models above, except for CodeLlama where the vector size is $4,096$. We set the token length to 512 for all language models during the tokenization process.

Generating Test Method Embeddings

A test method embedding is a numerical vector representation of a test method that captures semantic and contextual information from the source code. It is based on how and on what data a model was pre-trained, as follows.

- CodeBERT pre-trained code representations are based on a large corpus called CodeSearchNet [32], which contains both natural language (e.g., code comments) and source code across six programming languages. Its follow-up models (GraphCodeBERT and UniXcoder) leveraged data flow information during pre-training, which captures relationships between variables in the input code fragments and ASTs of the source code, respectively, to enhance the code representation.
- StarEncoder was pre-trained on a large dataset encompassing 86 programming languages collected from The Stack [37], which is specifically crawled from GitHub repositories for pre-training code LLMs.

- CodeLlama was pre-trained on a massive dataset (500***Billion*** tokens) containing both source code and natural language. CodeLlama has multiple variations and model sizes, depending on the type of training data and parameter size. In this chapter, considering the importance of scalability, we used the smallest version of CodeLlama (i.e., CodeLlama-7b), which has seven billion parameters.

In terms of model architecture:

- CodeBERT and StarEncoder employ a multi-layer bidirectional self-attentive Transformer [64] as model architecture to help the model capture contextual and positional information for each token from the entire input sequence.
- GraphCodeBERT extends such architecture using a graph-guided masked attention function to help the model encode graph-based data flow information.
- UniXcoder, on the other hand, is based on a multi-layer Transformer with a prefix denoting the pre-training mode of the model. The model architecture for the encoder-only mode, which is employed by LTM, allows the model to learn, for each token, the contextual information from the entire input sequence.
- CodeLlama uses an optimized auto-regressive transformer, enabling the model to efficiently predict missing parts within code sequences. Despite being a decoder-only model, CodeLlama exhibits the capability to understand the test code context and generate missing parts accurately.

During pre-training:

- CodeBERT employs Masked Language Modeling (MLM) [18], which allows the model to predict the masked tokens from the input sequence and whether a token was randomly replaced in a given sequence [13]. These two tasks help the model generate code embeddings that contain more accurate contextual information, accounting for the position of the tokens in the source code input.
- GraphCodeBERT also employs MLM with two additional tasks: Edge Prediction [28] and Node Alignment [28]. Edge Prediction allows the model to predict which edges, which denote dependencies between variables, have been masked for a given variable in the data flow graph. Node Alignment allows the model to predict which code token is related to a given variable in the data flow graph. These two tasks further enhance the code representation using data flow information.
- UniXcoder also uses the MLM task for the encoder-only mode, and further learns the test method embedding, which is first generated by taking the average (i.e., mean pooling) of all token embeddings and then further trained by utilizing multi-modal contrastive learning [24] and cross-modal generation [27], which both leverage AST and code comments to further enhance test method embeddings. Multi-modal contrastive learning generates different embeddings for the same input but with different

dropout masks, which allows the model to predict the original embedding using the other generated embeddings. This task helps the model to better distinguish between different method embeddings and thus can better deal with downstream tasks, such as test method similarity analysis [24]. Cross-modal generation generates code comments for the input test method. This helps the model to fuse the semantic information from the code comments, which describe the function of the code, to the test method embeddings.

- StarEncoder also employs MLM in addition to Next Sentence Prediction (NSP) [18], which helps the model learn the relationship between test code segments by predicting whether segments follow other segments.
- Unlike the above models, CodeLlama employs the code infilling task, which allows the model to generate the missing part of a code snippet while comprehending the entire context. This enables the model to understand the logical structure of the test code and the relationships between test code segments.

For LTM using CodeBERT, GraphCodeBERT, and StarEncoder, we use the embedding that corresponds to the $[CLS]$ token as a test method embedding, since it aggregates the information from all tokens of that method. For LTM using UniXcoder, we use the pre-trained test method embedding corresponding to each test method. For LTM using CodeLlama, we rely on the mean pooling of the last hidden states (i.e., the average of all token embeddings extracted from the last hidden layer of the model), which is also an effective strategy that aggregates the information from all code tokens [43].

3.3.2 Similarity Measurement of Test Method Embeddings

LTM employs two similarity measures for calculating the similarity between test method embeddings: Cosine Similarity and Euclidean Distance. They measure the similarity between test method embeddings from different aspects: Cosine similarity measures the angle between two vectors, whereas Euclidean distance calculates the straight-line distance between them.

Cosine similarity. This is a measure of similarity between two vectors based on the cosine of the angle between them [26], which is the dot product of the vectors divided by the product of their lengths, and is calculated as follows:

$$Cosine\ Similarity = \frac{\mathbf{T_1} \cdot \mathbf{T_2}}{\|\mathbf{T_1}\|\|\mathbf{T_2}\|} \tag{3.1}$$

where $\mathbf{T_1}$ and $\mathbf{T_2}$ denote the embeddings of test case T_1 and T_2, respectively.

The value of Cosine similarity ranges from -1 to 1. The higher the value of Cosine similarity, the more similar the two test cases are. In order to bound the value of Cosine similarity between 0 and 1, we normalized it as follows:

$$Norm.\ Cosine\ Sim. = 1 - \frac{\arccos \frac{\mathbf{T_1} \cdot \mathbf{T_2}}{\|\mathbf{T_1}\| \|\mathbf{T_2}\|}}{\pi} \tag{3.2}$$

Euclidean distance. This is a measure of similarity between two vectors based on the square root of the sum of squared differences between corresponding elements of the two vectors [26], which is calculated as follows:

$$Euclidean\ Distance = (\sum_{i}^{m} (t_{1i} - t_{2_i})^2)^{1/2} \tag{3.3}$$

where t_{1i} and t_{2i} is the i^{th} element of embedding T_1 and T_2, respectively. m is the total number of elements in each embedding ($4,096$ for CodeLlama and 768 for the other models).

The value of Euclidean Distance ranges from 0 to ∞. The higher the value of Euclidean distance, the less similar the two test cases. In order to bound the Euclidean distance between 0 and 1, and obtain a similarity measure, we normalized the distance as follows [19]:

$$Norm.\ Euclidean\ Dist. = \frac{1}{1 + (\sum_{i}^{m} (t_{1i} - t_{2_i})^2)^{1/2}} \tag{3.4}$$

We used the '`pdist`' function[1] from the *Scipy* Python library, which performs pairwise calculations for large datasets. The '`pdist`' employs highly optimized C code to improve the efficiency of similarity calculation on vector-based data, and stores the output in a condensed matrix, which further reduces the required memory resources.

3.3.3 Search-based Test Suite Minimization

Given that test suite minimization is an NP-hard problem [30], it can be addressed efficiently using meta-heuristic search algorithms [50] to find feasible, near-optimal solutions, a minimized test suite containing diverse test cases for a given budget in our context. Like ATM, we relied on a Genetic Algorithm (GA) since it has shown, in the context of TSM, to achieve a better trade-off between effectiveness and efficiency than its multi-objective alternative, namely Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [50]. The optimization problem addressed by the GA is defined as a fixed-size subset selection problem [8]. Each solution (chromosome) is a subset and is represented as a binary vector where 1 denotes the selection of the test case and 0 otherwise. The vector length equals the total number of test cases in the test suite before minimization. Given specified minimization budgets (25%, 50%, and 75%), the percentage of selected test cases in each solution is set to equal to the budget. For the fitness function, we used the summation of the maximum squared similarity values (i.e., Normalized Cosine Similarity and Normalized Euclidean Distance), shown below:

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html`

$$Fitness = \frac{\sum_{i,t_i \in M_n} (Max_{i,j,t_i,t_j \in M_n, i \neq j} \ Sim(t_i, t_j))^2}{n} \tag{3.5}$$

where M_n is a minimized test suite of size n, and $Sim(t_i, t_j)$ is the normalized similarity score of test cases pair t_i and t_j.

To select hyperparameter values, we followed the same published guidelines[2] as ATM for GA search, by using a population size of 100, a mutation rate of 0.01, and a crossover rate of 0.90. The GA process maximizes the diversity of the subset by iteratively evolving the population using crossover and mutation operators while evaluating each subset using the predefined fitness function. For each generation, the size of each solution (subset) remains fixed and equal to the minimization budget. This process is repeated until the fitness value improvement across generations is less than 0.0025.

Sim	*TC1*	*TC2*	*TC3*
TC1	0	0.8	0.3
TC2	0	0	0.6
TC3	0	0	0

Figure 3.2: An example of how similarity values of pairs of test cases are represented using a matrix. Values on and below the diagonal are set to 0 as they are either useless or duplicates.

In terms of implementation, LTM optimizes GA search by changing the data structure of the input (i.e., similarity values) to (1) accelerate the fitness calculations and (2) enhance memory usage. The data structure employed by ATM for storing similarity values is a data frame [3], which has a complex indexing and labeling mechanism, thus making fitness calculation computationally expensive. Therefore, we use matrices [4] instead, since they facilitate faster numerical calculations due to the fact that (1) it is a simpler data structure without complex indexing and labeling and (2) the matrix operations are implemented using highly optimized C code that ensures efficient computation and memory management. As shown in Figure 3.2, we stored the similarity values for all pairs of test cases in a matrix format without any duplicate and useless values (the similarity values on and below the diagonal of the matrix were set to 0). To save memory and further accelerate calculations, we employed a sparse matrix format [5], which exclusively preserves the value and position (indicating for which pair of test cases a given similarity value represents) of non-zero similarity values. Note that we did not parallelize the fitness calculation as it significantly increases the memory required by the matrix, which must contain similarity values for all considered subsets, making it less scalable. Although fitness calculations can be parallelized in other ways, such as multiprocessing, which utilizes multiple CPUs to

[2] `https://www.obitko.com/tutorials/genetic-algorithms/recommendations.php`
[3] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html
[4] https://numpy.org/doc/stable/reference/generated/numpy.array.html
[5] https://docs.scipy.org/doc/scipy/reference/sparse.html

parallelize fitness calculation, or distributed computing [8], which distributes the fitness calculation to various devices or nodes, they require additional computational resources, which contradicts our goal of improving scalability.

3.4 Validation

This section reports on the experiments we conducted to assess the effectiveness and efficiency of LTM.

3.4.1 Research Questions

RQ1: How does LTM perform for test suite minimization under different configurations? Test suite minimization aims to reduce the number of test cases in a test suite by removing redundant test cases while maintaining its fault detection power, thus achieving higher testing efficiency and effectiveness. In particular, the performance of LTM can be influenced by (a) the language models used for extracting test method embeddings and (b) the distance functions used for measuring the similarity between test method embeddings. In this RQ, we assess the performance, in terms of effectiveness and efficiency, of LTM under different configurations, each with a different combination of language model and distance function. We evaluated LTM using three minimization budgets (i.e., 25%, 50%, and 75%).

- ***RQ1.1: How effectively can LTM minimize test suites?*** The language models used for LTM (i.e., CodeBERT, GraphCodeBERT, UniXcoder, StarEncoder, and CodeLlama-7b) vary in terms of architectures and training data, thus generating embeddings that capture different information about test case code. The distance functions we used to measure the similarity or distance between test method embeddings differently, which can lead to different comparison results. Therefore, the combination of language models and distance functions can affect the ability of LTM to remove redundant test cases, which can in turn affect its fault detection capability. In this RQ, we assess the effectiveness of LTM in terms of fault detection capability under different configurations. In addition, we calculated the execution time of minimized test suites to quantify the achieved time savings resulting from test suite minimization.

- ***RQ1.2: How efficiently can LTM minimize test suites?*** Test suite minimization should run within practical time. The use of different combinations of language models and distance functions can impact the preparation time, and the resulting similarity values can guide the GA search to varying degrees of efficiency, thus resulting in different search time. In this RQ, we assess the efficiency of LTM in terms of preparation time (taken for test code tokenization, embeddings extraction, and similarity calculation) and search time (taken for GA search).

RQ2: How does LTM compare to ATM? ATM was developed as a black-box test suite minimization approach that achieved a better trade-off, in terms of effectiveness and efficiency, than the SOTA approach (FAST-R [15]). However, the minimization time of ATM increased rapidly with test suite size, making it less scalable for large projects with large test suites. For example, ATM required more than 10 hours, in terms of total minimization time, to minimize a large project in the dataset used for its evaluation, thus indicating scalability issues. In this RQ, we assess the performance of LTM compared to the two best configurations of ATM, specifically ATM using GA with combined and tree edit distance similarity measures, in terms of both effectiveness and efficiency.

- ***RQ2.1: How effectively can LTM minimize test suites compared to ATM?*** We aim to find an informative similarity measure that can better capture similarities between test cases, thus achieving higher *FDR*. In this RQ, we compare the best configuration of LTM to the two best configurations of ATM in terms of fault detection capability.

- ***RQ2.2: How efficiently can LTM minimize test suites compared to ATM?*** To achieve better scalability compared to ATM, we aim to reduce (a) preparation time, by investigating a similarity measure that is more efficient to calculate, and (b) search time, by investigating a more informative similarity measure that can better guide the search, thus resulting in faster convergence. In this RQ, we compare the efficiency of LTM to the two best configurations of ATM in terms of both preparation time and search time. Further, to better assess scalability for both LTM and ATM, we fit a regression model to investigate how minimization time (the sum of preparation time and search time) increases with the size of a test suite.

3.4.2 Experimental Design and Dataset

We conducted experiments to evaluate the performance of alternative LTM configurations, each with a different combination of language model and similarity measure, resulting in ten configurations in total (five different language models and two similarity measures), using the same experimental design and dataset as ATM. We conducted our experiments on a cluster of $1,340$ nodes with $83,216$ available CPU cores, each with a $2x$ AMD Rome 7532 with 2.40 GHz CPU, 256M cache L3, 249GB RAM, running CentOS 7. We considered each Java project version as an independent subject and conducted our experiments on all Java project versions (835 in total) in the dataset to increase the number of instances for experimental evaluation and, therefore, enhance the soundness of our conclusions. Each TSM approach was performed on every project version and ran 10 times to account for randomness and draw statistically valid conclusions, using three minimization budgets (25%, 50%, and 75%), which is the percentage of test cases that retained in the minimized test suite). There is a total of $25,050$ runs (835 Java project versions $\times$ 10 runs per version $\times$ 3 minimization budgets) for each configuration. Overall, the experiments, including similarity calculations, GA search, as well as executing all test suites 10 times, took more than three months in calendar time corresponding to 6 years of computation. We identified the

best configuration of LTM considering both the effectiveness and efficiency and compared it to the two best configurations of ATM. Our data, scripts, and raw results can be found in our replication package [52]

Baseline approach (ATM)

We compared LTM to ATM [50], a similarity-based, search-based test suite minimization approach that relies solely on the source code of test cases. ATM preprocesses test code by removing information that is irrelevant to testing logic from the test code, then converts test code into ASTs. ATM calculates four different tree-based similarity measurements between test case ASTs: top-down, bottom-up, combined and tree edit distance similarity. Note that ATM does not use the full code (i.e., the source code of test cases without preprocessing) as it would make the already-long preparation phase of ATM much longer and impractical in realistic settings. This is due to the fact that (1) the XML files used by ATM to store Abstract Syntax Trees (ASTs) of test cases would require substantial storage space and (2) it would take longer time to traverse the ASTs of test cases when calculating similarity, thus yielding much longer preparation time for transforming the test case code into ASTs and calculating similarity values. As a result, ATM's preparation time can take up to 41.2% of the total minimization time, and using the full code in ATM would make matters much worse.

After the preprocessing step, search algorithms (i.e., GA and NSGA-II) were then employed based on similarity values to find an optimal subset of test cases by removing redundant test cases. Using GA with combined similarity, ATM achieved a better trade-off in terms of effectiveness and efficiency compared to a SOTA test suite minimization approach, FAST-R [15]. However, it suffered from scalability issues as, for example, it took more than 10 hours to minimize $4k$ test cases for the largest test suite. This motivated us to compare LTM to ATM, in terms of fault detection capability and, more importantly, minimization time. We aim to assess whether LTM can be more scalable than ATM while achieving comparable or even higher fault detection capability. We compared LTM to the two best configurations of ATM, using GA with combined similarity (ATM/Combined) and tree edit distance (ATM/TreeEditDistance). The former was the best configuration considering the trade-off between effectiveness and efficiency, whereas the latter achieved higher fault detection capability but was more time-consuming than the former in terms of total minimization time. We further evaluated the achieved saving in testing time for every minimized test suite for both LTM and ATM. We relied on the publicly available replication package of ATM[6] provided by its authors.

Dataset

To facilitate comparisons, we ran our experiments using the same dataset as ATM to evaluate the effectiveness and, more importantly, the scalability of LTM. The dataset consists of 16 Java projects collected from Defects4J, a well-known dataset that offers

[6] https://doi.org/10.5281/zenodo.7455766

real and reproducible faults extracted from real-world, open-source Java projects to support software testing research. There is no publicly available industrial system or open source system that contains information to automatically trace test failures to system faults, a key requirement for our experiments. To further validate the performance of LTM, we included an additional large project from Defects4J, called ***Closure***, which was not part of the dataset used to evaluate ATM due to scalability issues. Each project has numerous faulty versions, each of which contains a single *real* fault that is present in the production code and leads to the failures of one or more test cases. It is worth noting that there is currently no publicly available dataset that offers multiple *real* faults per version, as automatically establishing a clear link between faults and test case failures would be challenging. Table 3.1 presents the characteristics of all 17 projects. Overall, the project sizes in terms of KLoc range from 2 to 179 KLoc, with the corresponding tests sizes ranging from 4 to 253 KLoc. The number of faulty versions ranges from 4 to 174 across projects, with the average number of test cases per version ranging from 152 to 7,308 across projects. Project sizes and tests sizes (i.e., lines of code) were extracted by analyzing the most recent version of each project using the *CLOC*[7] tool. In summary, we evaluated LTM using a total of 835 versions from 17 projects. However, given the scalability limitations of ATM [50], we compared our results to ATM using only the 661 versions from the original 16 projects, excluding the ***Closure*** project.

Table 3.1: Summary of the 17 Java projects

Project	Project Size (KLoC)	# of versions (faults)	Average # of test cases (methods) per version	Tests Size (KLoC)
Chart	92	26	1,817	41
Cli	2	39	256	4
Codec	9	18	413	15
Collections	30	4	1,040	38
Compress	45	47	404	29
Csv	2	16	193	7
Gson	9	18	984	20
JacksonCore	31	26	356	45
JacksonDatabind	74	112	1,814	72
JacksonXml	6	6	152	10
Jsoup	14	93	494	13
JxPath	20	22	250	6
Lang	30	64	1,796	61
Math	71	106	2,078	73
Mockito	21	38	1,182	36
Time	30	26	3,918	56
Closure	179	174	7,308	253

Evaluation metrics

We used the same evaluation metrics as ATM, specifically fault detection rate and total minimization time, in addition to the achieved saving in testing time of minimized test suites.

[7] `https://github.com/AlDanial/cloc`

Fault Detection Rate (FDR). We use *FDR* to assess the effectiveness of LTM compared to ATM. The *FDR* was calculated for each project as follows:

$$FDR = \frac{\sum_{i=1}^{m} f_i}{m} \tag{3.6}$$

where m refers to the total number of versions (system faults). For each version i, f_i equals to 1 if at least one failing (or fault-triggering) test case is included in the minimized test suite, or 0 otherwise.

Total Minimization Time (MT). We assess the efficiency of LTM compared to ATM by computing (1) the *preparation time*, which includes the time required for loading the language models, tokenizing test case code, extracting test method embeddings, and calculating similarity values between all pairs of test cases, and (2) the *search time*, which is the time required for running the GA.

Time Saving Rate (TSR). We use *TSR* to quantify the achieved testing time savings resulting from test suite minimization. It is important to note that the reduction in test execution time may not be directly proportional to the reduction in the test suite size, as the time required to execute individual test methods may vary [36]. Using the DEFECTS4J infrastructure, we ran the test suite for each project version (835 versions in total) 10 times and collected the average execution time for each test method in a clean, stable environment, a cluster with many available CPU cores. This ensures that each run runs independently and is not impacted by external interferences and fluctuations in memory or CPU performance. Then, we calculate the *TSR* for each minimized test suite as follows:

$$TSR = (1 - \frac{test\ execution\ time\ after\ TSM}{test\ execution\ time\ before\ TSM}) * 100 \tag{3.7}$$

Fisher's exact test. We use Fisher's exact test [55], a non-parametric statistical hypothesis test, to assess whether the difference in proportions of detected faults between the alternative LTM and ATM configurations is significant.

3.4.3 Results

In this section, given space constraints, we only report results for a minimization budget of 50% (the percentage of test cases retained in the minimized test suite) as trends and conclusions are similar for other budgets (25% and 75%), which are available in our replication package [52].

Using full test code versus preprocessed test code in LTM

One important question is whether the source code of test cases requires preprocessing, such as removing test case names, logging statements, comments, and test assertions. Though such preprocessing was performed by ATM [50], due to necessity (Section 3.4.2), this information may still convey relevant information about test case similarity, thus

retaining them can potentially improve fault detection capability. This information can also help language models better learn about semantic and contextual information about test cases and thus generate more informative test method embeddings. For example, code comment *'Tests the equals method'* describes the functionality of the test code, thus providing potentially important semantic information [27]. Therefore, we conducted a preliminary study to compare the *FDR* results of LTM using full code (i.e., the source code of test cases without preprocessing) and preprocessed code on a sample of 10 smaller projects from the DEFECTS4J dataset. We observed that LTM achieved higher *FDR* results using full code than using prepossessed code (+0.02 on average) across all configurations, thus making the former a better choice for our experiments. We summarize the results and provide further details in our replication package [52].

Improvement using optimized GA.

As discussed in Section 3.3.3, we optimized GA to accelerate the search process. Our results revealed that, compared to *MT* before optimization, the search time of ATM/Combined and ATM/TreeEditDistance is 224.66 and 203.02 times faster per project version, respectively. For LTM, the search time is 190.48 times faster per project version, across all configurations, which is a significant improvement in scalability. Note that the following comparisons between LTM and ATM are based on applying this optimized GA to both approaches.

RQ1 results

Tables 3.2, 3.3, and 3.4 provide the results of LTM in terms of *FDR*, *MT* (in min), and *TSR* (in percentage), respectively, when using CodeBERT, GraphCodeBERT, UniXcoder, StarEncoder, and CodeLlama-7b with Cosine and Euclidean similarity measures, for a 50% minimization budget.

RQ1.1 results. Table 3.2 shows that all LTM configurations achieved a high *FDR*, ranging from 0.65 to 0.95 across projects, with both the mean and median ranging from 0.75 to 0.84 for all configurations, for a 50% minimization budget. The highest overall average *FDR* was achieved using UniXcoder/Cosine (mean = 0.84, median = 0.82), ranging from 0.75 to 0.95 across projects. However, LTM using CodeBERT/Euclidean also yielded a high *FDR* (mean = 0.82, median = 0.84). CodeBERT/Euclidean and UniXcoder/Cosine showed standard deviations of 0.06 and 0.07, respectively. In addition, Fisher's exact test results reveal that LTM using UniXcoder/Cosine achieved significantly better *FDR* results than all other configurations, with $\alpha = 0.01$.

Figure 3.3 depicts the average *FDR* for all projects across generations, for all configurations of LTM. Compared to other configurations, LTM using UniXcoder/Cosine converges faster to a higher *FDR* (0.84 across projects), suggesting that it offers better guidance for the GA search, making it more effective at removing redundant test cases and thus leading to a better fault detection rate.

We observed that, for each language model, there seems to be a significant difference in *FDR* between using Cosine Similarity and Euclidean Distance, which is further confirmed by Fisher's exact test with $\alpha = 0.01$. This suggests that *FDR* results are influenced by the combination of language models and distance functions. The embeddings generated by UniXcoder and GraphCodeBERT are enriched with detailed information about the nodes and edges within test code ASTs. This could be the reason that Cosine Similarity, as a directional similarity, is more informative in distinguishing such test method embeddings than Euclidean Distance, a magnitude similarity. For CodeLlama-7b, the embedding dimensionality is $4,096$, compared to 768 for other language models, which can explain its better performance with Cosine Similarity over Euclidean Distance, since the latter is less effective in high-dimensional spaces [68].

In certain projects (*Cli*, *Codec*, *Csv*, and *Time*), UniXcoder/Cosine yielded a 5% to 10% lower *FDR* than other configurations. However, Fisher's Exact Test revealed that those differences are not significant. Nevertheless, we provide some tentative explanations regarding these projects: (1) A larger training data set covering a greater variety of programming languages (StarEncoder) and a substantially greater number of parameters (CodeLlama) might be beneficial for understanding test code from these projects, and (2) using ASTs as input for pre-training might introduce unnecessary details into test method embeddings, if test cases in these projects are similar in terms of code structure but rather differ regarding low-level details (e.g., complex string manipulation or domain-specific character strings), thus making UniXcoder less effective in understanding their test code.

In addition, we observed that UniXcoder/Cosine achieved a higher average *FDR* than CodeLlama-7b/Cosine (mean = 0.80, median = 0.81) and CodeLlama-7b/Euclidean (mean = 0.76, median = 0.76). This suggests that language models with fewer parameters (125*Million* for UniXcoder) can outperform much larger language models (7*Billion* for CodeLlama-7b) for our minimization problem. This can be attributed to the fact that CodeLlama-7b, like most large language models, is a decoder-only model that was pre-trained for code generation tasks, while UniXcoder employs code understanding tasks that are more effective in analyzing and comprehending the contextual semantic information from the source code [21].

Recall that the GA terminates when the fitness value improves by less than 0.0025, thus resulting in different numbers of generations for each configuration across project versions. The overall average number of generations of all configurations across versions is 47.73, with UniXcoder/Cosine having a higher average (63.10 generations). However, if a fixed number of generations had been used, say 40 generations, UniXcoder/Cosine would still yield the highest *FDR* (0.81) among all configurations. This clearly makes UniXcoder/Cosine the best LTM alternative, in terms of *FDR*.

Achieved Saving in Testing Time. Table 3.4 shows that, for the 50% minimization budget, LTM achieved an average *TSR* ranging from 41.15% to 43.37%, across configurations. The average *TSR* achieved by UniXcoder/Cosine was 41.72%. This indicates that, reducing the number of test cases by 50% results in a 41.72% reduction in testing time.

RQ1.2 results. Table 3.3 shows that the average *MT* of all LTM configurations ranges

Project \ Approach	CodeBERT		GraphCodeBERT		UniXcoder		StarEncoder		CodeLlama-7b		UniXcoder-p	ATM	ATM
	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Combined	Tree Edit Distance
Chart	0.82	0.84	0.79	0.75	0.82	0.69	0.71	0.79	0.72	0.73	0.77	0.88	**0.89**
Cli	0.81	0.81	0.83	0.86	0.82	0.81	0.83	0.86	**0.87**	0.86	0.80	**0.87**	**0.87**
Codec	0.89	**0.90**	0.83	0.88	0.80	0.81	0.82	0.83	0.87	0.78	0.78	0.77	0.80
Collections	0.70	0.82	0.95	0.75	0.95	0.70	0.70	0.75	0.78	0.70	**1.00**	0.98	0.92
Compress	0.85	0.86	0.87	0.87	0.86	0.81	0.78	0.84	0.85	0.79	**0.89**	0.88	**0.89**
Csv	0.86	0.87	0.88	0.91	0.85	0.79	0.84	0.86	**0.93**	0.88	0.88	0.92	0.91
Gson	0.70	0.71	0.72	0.75	**0.81**	0.75	0.76	0.76	**0.81**	0.77	0.76	0.76	0.77
JacksonCore	0.70	0.73	0.73	0.73	0.75	0.67	0.74	0.74	0.75	0.75	**0.78**	0.69	0.72
JacksonDatabind	0.66	0.69	0.68	0.67	**0.76**	0.65	0.65	0.69	0.67	0.65	0.74	0.69	0.70
JacksonXml	0.77	0.85	0.83	0.73	0.83	0.82	0.67	0.65	0.83	0.85	**0.87**	0.60	0.73
Jsoup	0.75	0.74	0.77	0.76	**0.80**	0.76	0.73	0.75	0.77	0.75	0.76	0.70	0.69
JxPath	0.88	0.90	0.92	0.86	**0.94**	0.83	0.72	0.77	0.78	0.75	**0.94**	0.79	0.79
Lang	0.81	0.85	0.86	0.75	**0.90**	0.71	0.77	0.85	0.83	0.71	0.84	0.78	0.83
Math	0.74	0.78	0.75	0.74	0.78	0.65	0.71	0.78	0.75	0.67	0.77	0.81	**0.82**
Mockito	0.80	0.83	0.84	0.81	**0.91**	0.79	0.77	0.83	0.80	0.77	0.78	0.77	0.79
Time	0.83	0.88	0.85	0.82	0.82	0.77	0.82	0.79	0.81	0.76	0.86	**0.89**	0.88
Statistics													
Min	0.66	0.69	0.68	0.67	**0.75**	0.65	0.65	0.65	0.67	0.65	0.74	0.60	0.69
25% Quantile	0.73	0.77	0.77	0.75	**0.80**	0.70	0.71	0.75	0.77	0.73	0.77	0.75	0.76
Mean	0.79	0.82	0.82	0.79	**0.84**	0.75	0.75	0.78	0.80	0.76	0.83	0.80	0.81
Median	0.81	**0.84**	0.83	0.76	0.82	0.77	0.75	0.79	0.81	0.76	0.79	0.79	0.81
75% Quantile	0.84	0.86	0.86	0.86	0.87	0.81	0.79	0.83	0.84	0.78	0.87	**0.88**	**0.88**
Max	0.89	0.90	0.95	0.91	0.95	0.83	0.84	0.86	0.93	0.88	**1.00**	0.98	0.92

Table 3.2: Results and descriptive statistics of *FDR* of LTM and ATM across projects for the 50% minimization budget. The highest *FDR* results are highlighted in bold.

Project \ Approach	CodeBERT		GraphCodeBERT		UniXcoder		StarEncoder		CodeLlama-7b		UniXcoder-p	ATM	ATM
	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Combined	Tree Edit Distance
Chart	**0.92**	1.51	0.97	1.20	1.33	0.99	2.24	2.68	21.02	20.92	2.18	7.86	94.22
Cli	**0.11**	0.16	0.14	0.15	0.17	0.14	0.16	0.22	4.39	4.38	0.22	0.23	1.20
Codec	**0.15**	0.21	0.18	0.17	0.22	0.18	0.25	0.33	5.75	5.65	0.31	0.42	3.02
Collections	**0.37**	0.51	0.43	0.50	0.58	0.38	0.79	1.11	8.96	8.80	0.87	2.16	14.89
Compress	**0.16**	0.20	0.19	0.18	0.22	0.17	0.25	0.34	9.26	9.18	0.32	0.55	6.55
Csv	**0.10**	0.13	0.12	0.11	0.15	0.11	0.14	0.16	4.19	4.15	0.19	0.16	0.35
Gson	**0.31**	0.39	**0.31**	0.33	0.47	0.32	0.68	0.79	8.31	8.26	0.77	1.48	6.09
JacksonCore	**0.14**	0.16	0.16	0.15	0.19	0.15	0.22	0.25	11.10	11.06	0.27	0.35	2.13
JacksonDatabind	0.92	1.40	**0.91**	1.35	1.43	1.33	2.15	2.46	34.67	35.41	2.50	4.90	22.69
JacksonXml	0.11	0.11	0.11	**0.10**	0.13	0.11	0.12	0.13	6.86	6.84	0.18	0.13	0.34
Jsoup	**0.17**	**0.17**	**0.17**	**0.17**	0.22	0.18	0.29	0.29	20.92	20.91	0.33	0.55	1.63
JxPath	**0.11**	0.15	0.13	0.13	0.17	0.13	0.17	0.21	4.98	4.95	0.23	0.22	0.73
Lang	**0.91**	1.47	**0.91**	1.08	1.38	0.97	2.35	2.82	15.09	14.98	2.53	5.44	33.47
Math	1.33	2.33	**1.31**	2.06	2.12	1.98	3.41	4.22	21.12	23.09	3.69	12.05	215.43
Mockito	**0.40**	0.50	0.42	**0.40**	0.59	0.41	0.97	1.12	17.40	17.93	1.15	2.32	9.34
Time	3.49	5.35	**3.43**	3.91	3.73	3.44	9.06	10.45	41.75	39.60	7.61	26.13	163.18
Statistics													
Min	**0.10**	0.11	0.11	**0.10**	0.13	0.11	0.12	0.13	4.19	4.15	0.18	0.13	0.34
25% Quantile	**0.13**	0.16	0.16	0.15	0.19	0.15	0.21	0.24	6.58	6.54	0.26	0.32	1.52
Mean	**0.61**	0.92	0.68	0.75	0.82	0.69	1.45	1.72	14.74	14.72	1.46	4.06	35.95
Median	**0.24**	0.30	0.25	0.26	0.35	0.25	0.49	0.57	10.18	10.12	0.55	1.02	6.32
75% Quantile	**0.91**	1.42	**0.91**	1.11	1.34	0.98	2.17	2.52	20.95	20.91	2.26	5.04	25.39
Max	3.49	5.35	**3.43**	3.91	3.73	3.44	9.06	10.45	41.75	39.60	7.61	26.13	215.43

Table 3.3: Results and descriptive statistics of MT (in min) of LTM and ATM across projects for the 50% minimization budget. The shortest MT results are highlighted in bold.

Project \ Approach	CodeBERT		GraphCodeBERT		UniXcoder		StarEncoder		CodeLlama-7b		UniXcoder-p	ATM	ATM
	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Combined	Tree Edit Distance
Chart	55.45%	57.35%	58.78%	59.81%	58.03%	56.11%	58.47%	**63.08%**	56.64%	59.14%	55.27%	59.77%	57.53%
Cli	29.90%	27.90%	**30.54%**	29.75%	28.32%	28.17%	29.52%	27.52%	28.25%	28.46%	29.95%	28.07%	27.34%
Codec	44.11%	40.68%	47.59%	49.93%	**55.58%**	50.62%	51.05%	48.38%	45.27%	48.83%	45.84%	34.62%	45.94%
Collections	57.40%	55.05%	56.21%	56.10%	55.41%	59.05%	**59.41%**	57.36%	55.01%	56.78%	56.37%	53.57%	57.65%
Compress	37.74%	37.98%	36.56%	38.10%	39.10%	**40.70%**	37.87%	35.26%	36.45%	40.11%	36.33%	33.85%	32.59%
Csv	18.66%	18.78%	19.90%	20.07%	18.13%	10.11%	16.40%	16.15%	12.82%	9.37%	6.92%	**20.45%**	19.89%
Gson	41.81%	40.60%	42.16%	42.01%	41.40%	**44.63%**	41.85%	40.10%	41.69%	43.63%	39.51%	40.35%	38.17%
JacksonCore	47.45%	**48.44%**	44.37%	46.18%	47.10%	48.00%	46.02%	46.42%	46.95%	46.98%	45.22%	43.59%	46.71%
JacksonDatabind	42.54%	41.28%	41.68%	41.63%	38.32%	42.37%	41.71%	39.56%	41.29%	41.75%	39.98%	**50.61%**	47.92%
JacksonXml	42.81%	42.05%	42.63%	44.12%	45.80%	46.32%	45.70%	**49.52%**	44.03%	43.60%	44.48%	39.17%	39.34%
Jsoup	42.96%	42.03%	41.33%	**43.80%**	33.90%	41.65%	38.60%	35.61%	35.12%	37.55%	32.55%	40.39%	38.20%
JxPath	43.37%	42.82%	41.00%	41.90%	40.83%	42.70%	42.46%	41.84%	44.12%	44.16%	41.04%	44.40%	**44.46%**
Lang	38.36%	36.49%	40.14%	41.24%	34.05%	44.76%	41.04%	35.37%	42.83%	**44.97%**	32.19%	37.71%	38.06%
Math	50.24%	49.43%	50.26%	49.47%	49.67%	49.67%	50.00%	49.72%	50.90%	**51.34%**	45.44%	47.93%	49.10%
Mockito	27.65%	29.87%	34.33%	38.38%	33.97%	39.93%	**44.82%**	41.80%	32.13%	39.77%	24.12%	24.31%	26.13%
Time	48.12%	47.57%	48.33%	48.49%	47.89%	**49.06%**	47.35%	46.94%	46.84%	48.38%	46.17%	45.93%	47.35%
Statistics													
Min	18.66%	18.78%	19.90%	20.07%	18.13%	10.11%	16.40%	16.15%	12.82%	9.37%	6.92%	**20.45%**	19.89%
25% Quantile	38.21%	37.61%	39.25%	40.53%	34.03%	**41.41%**	40.43%	35.55%	36.12%	40.03%	32.46%	34.43%	36.69%
Mean	41.79%	41.15%	42.24%	43.19%	41.72%	**43.37%**	43.27%	42.16%	41.27%	42.80%	38.84%	40.30%	41.02%
Median	42.89%	41.66%	41.92%	42.91%	41.12%	**44.70%**	43.64%	41.82%	43.43%	43.90%	40.51%	40.37%	41.90%
75% Quantile	47.62%	47.79%	47.78%	48.74%	48.34%	**49.21%**	48.01%	48.67%	46.87%	48.49%	45.54%	46.43%	47.49%
Max	57.40%	57.35%	58.78%	59.81%	58.03%	59.05%	59.41%	**63.08%**	56.64%	59.14%	56.37%	59.77%	57.65%

Table 3.4: Results and descriptive statistics of *TSR* (in percentage) of LTM and ATM across projects for the 50% minimization budget. The highest *TSR* results are highlighted in bold.

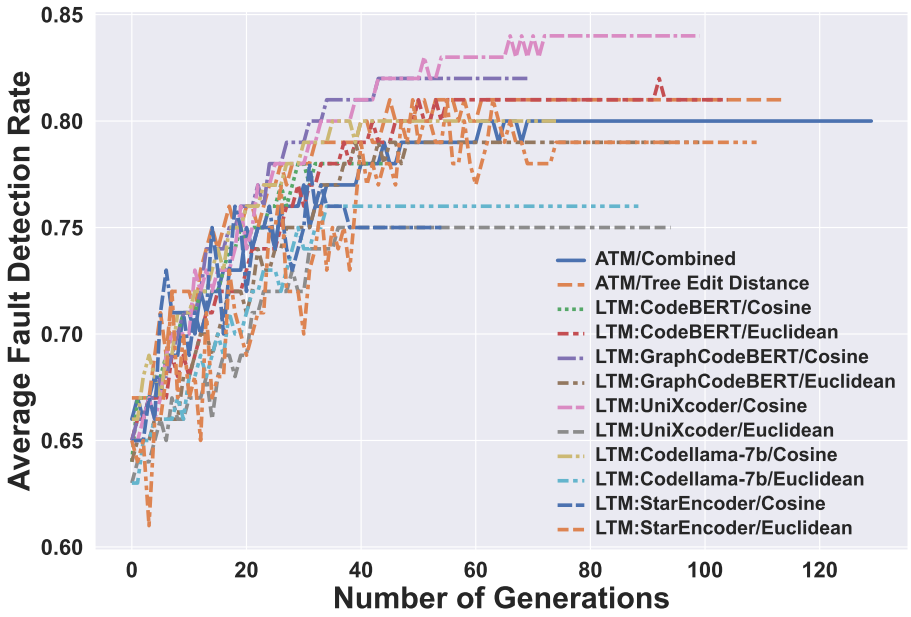


Figure 3.3: *FDR* across projects for each generation of LTM and ATM

Table 3.5: Results of *FDR* (in percentage), *MT* (in min) and *TSR* (in percentage) of LTM for the ***Closure*** project and a 50% minimization budget. The highest *FDR* and *TSR* as well as the shortest *MT* are highlighted in bold.

Metric \ Approach	CodeBERT		GraphCodeBERT		UniXcoder		StarEncoder		CodeLlama-7b	
	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean	Cosine	Euclidean
FDR	0.77	0.77	0.78	0.78	**0.79**	0.74	0.73	0.76	0.77	0.76
MT	**15.02**	30.53	19.73	25.98	17.80	27.61	16.45	34.17	54.87	66.84
TSR	53.18%	51.98%	53.75%	52.53%	52.55%	54.27%	**54.40%**	52.51%	53.81%	53.70%

from 0.61 to 14.74 min per project version. The *MT* of LTM using UniXcoder/Cosine, which achieved the highest average *FDR* (0.84), also appears to be efficient with $mean = 0.82\ min$ and $median = 0.35\ min$.

Preparation Time. The average preparation time ranges from 0.26 to 17.71 min per project version, across configurations. We observed that the preparation time of UniXcoder/Cosine is 65.43 times shorter than CodeLlama-7b. This is because (1) larger language models require much longer model loading time (12.07 min for CodeLlama-7b, compared to UniXcoder for 0.04 min, on average) and significantly larger memory (13.10GB for CodeLlama-7b compared to 504MB for UniXcoder) and (2) the embeddings generation time of larger language models is significantly longer (5.15 min for CodeLlama-7b compared to 0.16 min for UniXcoder, on average) as they have more complex architectures and a larger number of parameters, making the generation of embeddings more time-consuming.

Search Time. The average search time of LTM ranges from 0.46 to 1.90 min per project version, across configurations. LTM using CodeBERT with Cosine similarity was the fastest in terms of search time, ranging from 0.04 to 3.86 min, across project versions. Not only LTM using UniXcoder/Cosine took a shorter search time (0.78 min) than the average search time across configurations (1.03 min), it also achieved the highest *FDR* among all configurations, thus making it the best option for LTM.

Results for the Closure project. We further evaluated LTM on the *Closure* project, which has 174 versions with an average of 7,308 test cases per version. Though this project is part of the DEFECTS4J dataset, it was previously excluded from the ATM evaluation due to its scalability limitations [50] and the large size of our experiments. However, evaluating the effectiveness and efficiency of LTM on this project, considering its larger scale and test suite, can be informative. Table 3.5 presents the results regarding *FDR*, *MT* (in min), and *TSR* (in percentage), respectively. We can see that UniXcoder/-Cosine achieves once again the highest *FDR* of 0.79 and an *MT* of only 17.80 ***min***, while saving 52.55% testing time, which indicates that LTM can be effective and scalable for larger projects and test suites.

RQ1 summary. LTM achieved high *FDR* results (0.79, on average, across configurations) for a 50% minimization budget. UniXcoder/Cosine is the best LTM configuration when considering both effectiveness (0.84 *FDR* on average) and efficiency (0.82 min on average), with an average *TSR* of 41.72%.

RQ2 results

RQ2.1 results. For a 50% minimization budget, Tables 3.2, 3.3, and 3.4 report *FDR*, *MT*, and *TSR*, respectively, for the best configuration of LTM (i.e., UniXcoder/Cosine) and the two best configurations of the baseline approach: ATM with combined similarity and tree edit distance similarity.

We observe that LTM significantly outperforms ATM/Combined, the best ATM alternative, and ATM/TreeEditDistance in terms of *FDR*, with $\alpha = 0.01$. Compared to ATM, LTM achieves higher average *FDR* (0.84) with lower standard deviation (0.06 compared to 0.10 for ATM/Combined and 0.08 for ATM/TreeEditDistance).

Figure 3.3 shows that UniXcoder/Cosine converges faster to a higher *FDR* (0.84 across projects), compared to ATM/Combined (0.80 across projects) and ATM/TreeEditDistance (0.81 across projects). This may be explained by the fact that (1) UniXcoder can handle full code (i.e., the source code of test cases without preprocessing), whereas ATM uses preprocessed code; and (2) UniXcoder was pre-trained on various code understanding tasks, enabling the model to learn contextual information for each token and the meaning of each code element (e.g., method, variable names, and string characters) in the input test code. In contrast, ATM relies on tree-based similarity based on ASTs, which lack information regarding the syntactic context and certain details of the test code.

For some projects, ATM achieved higher *FDR* than LTM. However, a Fisher's Exact Test suggests that, for these projects, there is no significant difference between the *FDR* results of ATM and LTM. Nevertheless, a possible explanation for such performance difference could be that the percentage of test code exceeding the token limits of language models on these projects (5.72%, 3.21%, 3.25%, and 7.25% for ***Chart***, ***Compress***, ***Collections***, and ***Math***, respectively) is higher than the average of 2.98% across projects, and that the truncated parts are important for distinguishing test cases. Another possible

explanation could be due to variability across test cases in some projects (e.g., ***Cli*** and ***Time***), regarding their low-level code elements such as complex string manipulation and domain-specific character strings, rather than structural information. This makes tree-based similarity better at capturing these differences, by comparing AST node labels, than using the Cosine Similarity of embeddings.

Note that the way ATM preprocessed test cases, by removing all the comments and assertions, may also influence its performance. For example, for the ***Codec*** project, 26.18% of the test methods contain comments, which is higher than the average percentage across projects (14.85%). Such comments may convey rich information about test code, such as test case descriptions and expected results. Therefore, due to such loss of information in code comments, ATM may not efficiently achieve higher *FDR* for this project.

The average number of generations required by LTM was 63.10 across project versions (maximum = 100), which is lower than ATM/Combined (mean = 85.60 and maximum = 130) and ATM/TreeEditDistance (mean = 74.88 and maximum = 115). This indicates that LTM converges faster across generations than ATM, in terms of fitness value, which is the sum of the maximum squared similarity values for all test case pairs per version. Related to this, if due to time constraints a fixed number of generations is used as the termination criterion for search, say 40 generations, it is clear based on Figure 3.3 that UniXcoder/Cosine would yield the highest *FDR*. This indicates that LTM offers better guidance for the GA search, making it more effective at removing redundant test cases while maintaining a higher fault detection rate, thus explaining the results in Table 3.2 and why LTM is a better option than ATM for test suite minimization in practice.

Achieved Saving in Testing Time. The average time saving rate of the best configuration of LTM ranges from 18.13% to 58.03% across projects, with a mean of 41.72% across projects. We observe that this time saving rate is slightly higher than, but not significantly different from, that of ATM/Combined (40.30%) and ATM/TreeEditDistance (41.02%), as shown by the Wilcoxon signed-rank test.

Comparing LTM and ATM using preprocessed code. As discussed in Section 3.4.2, ATM cannot handle full code due to scalability issues. However, the preprocessing step might remove useful information from the test code and thus impact the effectiveness of ATM. Therefore, in order to explain the LTM *FDR* results when compared to ATM, we want to determine how the use of full code and the LTM test method embeddings individually affect *FDR*. For this, we conducted an additional experiment to assess LTM with UniXcoder/Cosine, the best LTM configuration, using preprocessed code, on all 16 projects. Tables 3.2 and 3.4 report the *FDR* and *TSR* results, respectively, for this configuration using preprocessed code (UniXcoder-p) and the two best configurations of ATM (using GA with combined and tree edit distance similarity measures), for a 50% minimization budget. We observe that, using the same input as ATM (preprocessed test code), LTM with UniXcoder/Cosine still yields a higher average *FDR* (0.83) than ATM (0.81). Still, it is lower than what we achieved using LTM with full code (0.84). The average *TSR* result for LTM with preprocessed code is however slightly lower than that of ATM (38.84% versus 41.02%), though the difference is not statistically significant, as resulted by the Wilcoxon signed-rank test. These results therefore suggest that the small improve-

ments in *FDR* observed with LTM are due to both LTM's test method embeddings and its capacity to handle full code in a scalable way. *TSR* remains similar for LTM and ATM, with or without preprocessing. We report the results for LTM with UniXcoder/Cosine using preprocessed code on all 16 projects in our replication package [52].

RQ2.2 results.

Figure 3.4 depicts *MT*, preparation time, and search time—the first one including the last two—as a function of the number of test cases per version, for the 661 versions of the 16 projects. This is done for the best configuration of LTM (i.e., UniXcoder/Cosine) and compared to the best configurations of ATM (ATM/Combined and ATM/TreeEditDistance) for the 50% minimization budget.

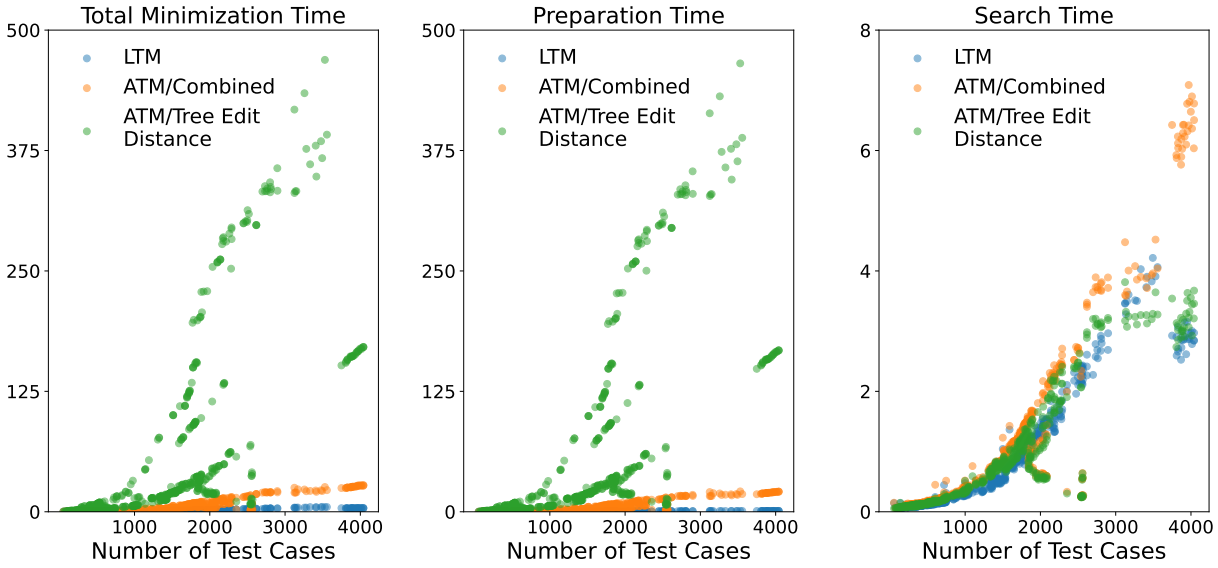


Figure 3.4: Scatter plots of the number of test cases and *MT*, preparation time, and search time (in min), for LTM (UniXcoder/Cosine) and ATM, across all the 661 project versions for the 50% minimization budget

We observe that LTM ran much faster than ATM in terms of *MT* (79.80% and 97.72% faster than ATM/Combined and ATM/TreeEditDistance on average, respectively) with both much less preparation time (92.70% and 99.35% less than ATM/Combined and ATM/TreeEditDistance on average, respectively) and search time (35.16% and 7.81% less than ATM/Combined and ATM/TreeEditDistance on average, respectively). In addition, we observe that the observations in Figure 3.4 follow a quadratic relationship rather than a linear one. Therefore, a quadratic regression model is fitted for the preparation time, search time, and *MT* as a function of the number of test cases per version, respectively, to quantify and compare relationships across approaches.

The equation of the quadratic regression model is as follows.

$$Time = a * n^2 + b * n + c \tag{3.8}$$

where n denotes the number of test cases per project version, a and b are regression coefficients, and c denotes the intercept of the quadratic regression model.

The results of quadratic regression confirmed that the MT and the number of test cases, for both LTM and ATM, follow a quadratic relationship as the coefficients on the quadratic term were statistically significant with $\alpha = 0.01$. This is, as expected, due to the fact that (1) similarity values are measured between test case pairs, which increases quadratically with the number of test cases, and (2) the search space for GA increases rapidly with the number of test cases. The R^2 values for the regression model of LTM and ATM/Combined were 98.89% and 89.51%, respectively. This indicates that a high percentage of the variation in MT can be explained by the number of test cases. However, the R^2 value for ATM/TreeEditDistance was only 50.36%, indicating that a much smaller proportion of the variation in MT can be explained by the number of test cases in this case. This is because the preparation time of ATM/TreeEditDistance, which is part of its MT, was also largely influenced by AST sizes. More discussions will be provided in the following section.

Moreover, a comparison of regression coefficients indicates that the MT of LTM increases up to two orders of magnitude slower than ATM with the squared number of test cases per project version: $8.255e^{-06}$ for LTM versus $9.314e^{-05}$ and $4.446e^{-04}$ for ATM/-Combined and ATM/TreeEditDistance, respectively. For large industry projects with large test suites, LTM is thus much more scalable in terms of MT, and therefore a better option than ATM in practice.

Preparation Time. We found that LTM is much more efficient, in terms of preparation time across project versions (mean = 0.23 min and median = 0.14 min), compared to ATM/Combined (mean = 3.15 min and median = 0.74 min) and ATM/TreeEditDistance (mean = 35.31 min and median = 6.10 min). Moreover, the preparation time of LTM takes only 35.75% of the MT per version, which is much less than ATM/Combined (71.22%) and ATM/TreeEditDistance (93.74%). Quadratic regression shows that the coefficient on the quadratic term for LTM ($1.241e^{-06}$) is one order of magnitude smaller than the coefficients for ATM/Combined ($6.634e^{-05}$) and two orders of magnitude smaller than ATM/TreeEditDistance ($4.3680e^{-04}$), with all coefficients statistically significant with $\alpha = 0.01$. This indicates that, on larger industry projects with much larger test suites, the absolute calculation time difference between LTM and ATM is expected to be extremely large, thus making LTM a better option regarding scalability in preparation time.

The R^2 values for the regression model of LTM and ATM/Combined are 97.70% and 87.86%, respectively. This indicates that more than 87% of the variation in preparation time can be explained by the number of test cases. However, the R^2 value for ATM/TreeEditDistance is 49.96%, which is mainly due to its preparation time as it is highly influenced by test code length. For instance, as shown in Figure 3.4, there are project versions with nearly $4k$ test cases from the ***Time*** project, which have less preparation time than project versions with around $3k$ test case from the ***Math*** project. This is due to the fact that the average test code for the ***Math*** project is longer than that for the ***Time*** project, resulting in its AST tree size being twice as long. Therefore, although the test suite size of the ***Math*** project is smaller, its preparation time is significantly longer than that of the ***Time***

project. In contrast, the preparation time for LTM is not affected by the length of the test code, which further supports our claims regarding its scalability.

Search Time. We observe that the time LTM takes for searching the optimal test suite (mean = 0.59 min and median = 0.21 min across projects) is also shorter than that for ATM/Combined (mean = 0.91 min and median = 0.29 min) and ATM/TreeEditDistance (mean = 0.64 min and median = 0.23).

Moreover, quadratic regression reveals that, while quadratically increasing with the number of test cases per version, the search time of LTM increases more slowly with the squared number of test cases per version (a coefficient on the quadratic term equal to $7.013e^{-06}$) than ATM/TreeEditDistance ($7.844e^{-06}$) and one order of magnitude more slowly than ATM/Combined ($2.680e^{-05}$), with more than 79% (R^2) of the variation in MT being explained by the number of test cases.

Note that the search time not only depends on the number of test cases per version but is also impacted by the number of generations required by the GA to converge. For instance, as shown in Figure 3.4, though the number of test cases per version for the *Math* project (2,078) is less than that for the *Time* project (3,918), GA needed more generations to converge for *Math*, thus resulting in longer search time. Test cases of the *Math* project are more diverse than that of the *Time* project, with a higher standard deviation of similarity values (0.13 compared to 0.09), thus leading to more generations and longer convergence time for the GA search.

In conclusion, in addition to being more efficient than ATM in terms of preparation time, the similarity measure employed by LTM provides better guidance and thus helps the search algorithm converge faster than ATM, thus resulting in less search time.

Cost Analysis To demonstrate the cost-effectiveness of LTM, we performed a cost analysis and identified the conditions under which the best configuration of LTM, UniXcoder/Cosine, becomes beneficial.

The costs associated with minimization and test execution are defined as follows:

- **Minimization Cost:** The cost of applying the LTM to a test suite containing m test cases, under r minimization budget, is formulated as:

$$C = a * m^2 \tag{3.9}$$

  where a is a constant derived from the quadratic regression model.

- **Execution Cost Before Minimization:** The cost of executing the test suite with m test cases is denoted as $Cost(m)$. Before minimization, the total execution cost of the test suite that is executed across N versions is:

$$\text{Total Execution Cost (before minimization)} = N \cdot \text{Cost}(m) \tag{3.10}$$

- **Execution Cost After Minimization:** With a minimization budget r, the minimized test suite contains $r \cdot m$ test cases. The execution cost after minimization becomes:

$$\text{Total Execution Cost (after minimization)} = C + r \cdot \text{Cost}(m) \cdot N \tag{3.11}$$

The difference in cost between before and after minimization is:

$$\Delta C = (1 - r) \cdot \text{Cost}(m) \cdot N - C. \tag{3.12}$$

For the LTM to be cost-effective, it must satisfy $\Delta C > 0$, leading to:

$$(1 - r) \cdot \text{Cost}(m) \cdot N > C \tag{3.13}$$

Rearranging for N, the minimum number of test suite executions required for cost-effectiveness is shown as follows:

$$N > \frac{C}{(1 - r) \cdot \text{Cost}(m)} \tag{3.14}$$

This implies that if N is sufficiently large—meaning the minimized test suite is executed a sufficient number of times—the cost after minimization will be lower than the cost before minimization.

For example, for the largest test suite from the *Time* project, the cost analysis is as follows:

- $m = 3918$ test cases,
- Minimization cost $C = 8.255 \cdot 10^{-6} \cdot m^2 = 126.72\,\text{s}$,
- Execution cost of m test cases $\text{Cost}(m) = 78.4\,\text{s}$,
- Minimization budget $r = 0.5$ (50%).

The minimum number of executions N is:

$$N > \frac{126.72}{(1 - 0.5) \cdot 78.4} = \frac{126.72}{0.5 \cdot 78.4} = 3.23. \tag{3.15}$$

Thus, if the minimized test suite is executed more than 4 times ($N > 4$), LTM is cost-effective.

For the test suite from the representative project, *Collections*, which has approximately the average number of test cases across all projects (i.e., 1040), the cost analysis is as follows:

- $M = 1040$ test cases,
- Minimization cost $C = 8.255 \cdot 10^{-6} \cdot m^2 = 8.93\,\text{s}$,
- Execution cost of m test cases $\text{Cost}(m) = 33.07\,\text{s}$,
- Minimization budget $r = 0.5$ (50%).

The minimum number of executions N is:

$$N > \frac{8.93}{(1 - 0.5) \cdot 33.07} = \frac{8.93}{0.5 \cdot 33.07} = 0.54. \tag{3.16}$$

Therefore, for the representative test suite with the average number of test cases across projects, the minimized test suite only needs to be executed more than once ($N > 1$) to make LTM cost-effective.

RQ2 summary. LTM outperforms ATM by achieving significantly higher *FDR* (0.84 versus 0.81, on average). However, the main benefit of LTM is that it is running much faster than ATM (0.82 min versus 4.06 min, on average), in terms of both preparation time (up to two orders of magnitude faster) and search time (up to one order of magnitude faster). Given the acute scalability issues met in industrial contexts regarding minimization for large test suites and systems, this result is of high practical importance.

3.4.4 Discussion

Scalable test suite minimization with much less preparation time than the SOTA.

Our results show that LTM runs up to two orders of magnitude faster than the SOTA approach (i.e., ATM) while achieving a statistically significantly higher fault detection rate. We should note that, according to the average *MT* across projects, LTM runs five times faster than ATM/Combined and 44 times faster than ATM/TreeEditDistance. These differences tend to increase significantly as the test suite size grows. For example, for the *Math* project with 2078 test cases, LTM runs 6 and 102 times faster than ATM/Combined and ATM/TreeEditDistance, respectively. This suggests that LTM, while achieving higher effectiveness, can save enormous time and resources for minimizing test suites of the much larger software systems and test suites encountered in practice. The preparation phase plays a crucial role in the scalability of LTM. Since we employed pre-trained language models to extract vector-based test method embeddings and used a highly optimized function to compute similarity between them, we managed to achieve much shorter preparation times compared to ATM. In addition, since we used the test code as input

without preprocessing it and converting it into other formats, such as ASTs for ATM, we saved substantial memory space as well.

Unlike coverage-based techniques, the size of a project (i.e., source lines of code) has no impact on the minimization time of black-box TSM approaches, including LTM and ATM, as minimization time mostly depends on the number of test cases per project version. In addition, we optimize the search process by changing the data structure for input similarity values and accelerating the fitness calculation, which reduced the search time by 190.48 times. The *Closure* project—the largest by far, comprising 179 KLoC and 174 versions, with 7,308 test cases per version—took only 17.80 min in terms of *MT* and achieved a *FDR* of 0.79, for a 50% minimization budget. Note that for ATM, *Closure* was far too large to even run our experiments, thus further illustrating the scalability gains with LTM.

Though LTM uses the same search termination criterion as ATM, which is defined as a fitness improvement of less than 0.0025 across generations, we observed that *FDR* reaches a plateau after a certain number of generations (e.g., about an average of 65 generations across project versions using LTM with UniXcoder/Cosine). However, the GA search continues as long as there is sufficient improvement in fitness (e.g., LTM with UniXcoder/-Cosine reached a maximum of 100 generations). This suggests that the search termination criterion unnecessarily prolongs the search and thus needs further investigation. For example, using a fixed number of generations, determined based on experience with past versions, could save time and resources while achieving high *FDR* values.

Effective test suite minimization reduces testing cost.

Further, LTM achieves an average saving of 41.72% in test execution time with a 50% test suite minimization budget. However, though this is due to the relatively modest size of the systems we experiment with, one might point out that the average execution time of test suites in our dataset is smaller than LTM minimization time. Nevertheless, LTM still offers significant advantages as testing is performed frequently, especially in CI contexts, while minimization is occasional, unlike test case selection and prioritization. It can thus achieve substantial savings in time and resources over time, particularly in CI environments, where numerous builds are tested for every code change, thus warranting the elimination of redundant or unnecessary test cases to significantly reduce the overall test execution time and required testing resources (e.g., memory, CPU). According to our cost analysis, after minimization with a 50% budget, the largest test suite only needs to be executed more than 4 times to provide a payoff in execution time. Additionally, test execution times tend to be significantly larger in industrial systems, thus making minimization even more beneficial.

More informative similarity measures that better guide the search.

Our results show that the similarity measures LTM employed better guide the GA to converge faster to a higher *FDR*, when compared to ATM, confirming the usefulness of using pre-trained language models for extracting test method embeddings.

Results also suggest that the *FDR* results are influenced by multiple factors including the language models, distance functions, and test code characteristics. Language models differ in the types of input used for pre-training, the model architectures employed, and the kinds of tasks they were pre-trained for, thus generating embeddings with various characteristics. While Cosine Similarity captures directional similarity by measuring the angle between embeddings [60], Euclidean Distance captures magnitude similarity by measuring the straight-line distance between embeddings in a multidimensional space [19]. Therefore, using different combinations of language models and distance functions is expected to yield varying *FDR* results.

In addition, the test code characteristic (e.g., domain-specific character strings or complex string manipulations) of certain projects may also impact *FDR* performance, depending on the input (test code or ASTs) and dataset (size and diversity of programming languages) used for pre-training the language models. In conclusion, the performance of LTM in terms of *FDR* may be affected by several confounding factors. Future research should explore the effects of these factors through deeper analyses of test code structures and controlled experiments.

Language models with fewer parameters may offer a better trade-off than those with many more parameters in TSM contexts.

The results show that UniXcoder (125*Million* parameter size) outperforms CodeLlama-7b (7*Billion* parameter size) in terms of both effectiveness and efficiency. This suggests that larger language models, though comprising substantially more parameters and being pre-trained on larger datasets, may not necessarily perform better for certain downstream tasks such as TSM. CodeLlama-7b, as a decoder-only model, was pre-trained for code generation tasks, such as code infilling. While generating (infilling) the next token, the structure of the decoder makes each token only learn information from preceding tokens, thereby potentially making its embedding less informative due to the absence of information from subsequent tokens. In contrast, code understanding tasks (e.g., MLM) allow the model to learn the information before and after each token, thus making it more effective in learning the contextual semantic information from the source code. Furthermore, UniXcoder enhanced the code representation by utilizing AST as input, which further enables the model to understand the source code structure and grammar, thus resulting in better *FDR*.

Note that, to the best of our knowledge, there is no publicly available open-source language model that outperforms UniXcoder on the code clone detection task, which is similar in some ways to our minimization task. More importantly, given that scalability is crucial for LTM, larger language models may require much more memory, computational resources, and time when generating embeddings due to their substantially higher number of parameters and complex model architectures. Therefore, for a downstream task, such as TSM, that requires assessing test code similarity, relatively smaller language models can be a better option than larger ones.

Using full code versus preprocessed code.

We recommend that LTM be used with the source code of test cases without preprocessing, since it yielded better results than when using preprocessed code, as reported in Section 3.4.3. However, some aspects of preprocessing, such as normalizing variable identifiers, may be beneficial, since they do not have an impact on the data flow in the source code. Therefore, using such techniques may potentially help language models generate embeddings that are more generalizable and suitable for code similarity calculation across projects, thus improving the fault detection power of LTM.

3.5 Threats to validity

Internal Validity Internal threats to validity are concerned with the ability to draw conclusions from our experimental results. To make fair comparisons and draw valid conclusions, we conducted our experiments using the same experimental design and dataset used to evaluate ATM [50]. We evaluated the alternative LTM configurations and compared the best configuration to the two best configurations of ATM. For minimizing test suites, we used the same fitness function and parameter settings as ATM, but we further optimized GA by utilizing a more efficient data structure for storing the input similarity values and accelerating the fitness calculation. We compared the performance using optimized GA for both LTM and ATM.

ATM used preprocessed code as input because of its very long preparation time. In contrast, LTM used full source code, raising the question of whether differences in *FDR* between LTM and ATM are due to differences in test representation (test method embeddings or ASTs) or code input (full code or preprocessed code). To answer that question, we conducted an additional experiment to assess the best configuration of LTM (i.e., UniXcoder/Cosine) using preprocessed code, which showed that LTM with preprocessed code still fares better than ATM in terms of *FDR*.

Another threat is related to the token length limit of language models, which might truncate the source code of some test cases [22], thus resulting in information loss that might in turn negatively impact the performance of LTM. However, we observed that, for each version, the percentage of test methods with the token length exceeding the limit is only 2.98% across projects, which can be considered negligible. Nevertheless, some projects have a relatively higher percentage of test methods exceeding that limit (e.g., 7.25% for *Math*), which might explain the slightly lower *FDR* of LTM compared to ATM on these projects. Future research should further investigate the use of additional language models with longer token length limits, though they might greatly impact efficiency.

External Validity External threats are concerned with the ability to generalize our results. We assessed both the performance of ATM and LTM on a large dataset consisting of 16 Java projects with a total of 661 versions collected from Defects4J for which we could trace failures to faults. We further validated the performance of LTM using an additional larger project from Defects4J, called *Closure*, consisting of 174 versions, for which

we could not evaluate ATM. Though we did not evaluate LTM on projects written in other programming languages, LTM can be easily adapted to other programming languages as the language models we used were pre-trained on multiple programming languages and have been demonstrated to yield good performance in various downstream tasks in other programming languages [23, 27, 28, 41]. Future research should extend our study to investigate LTM on projects with other programming languages to further generalize our conclusions. In addition, test method execution time may vary from run to run. To measure the achieved time-saving in test execution time more precisely, we used the DEFECTS4J infrastructure to run test suites 10 times and collect the average execution time for each test method in a clean, stable environment, which ensures we run each test suite without interference and fluctuations in memory or CPU usage (e.g., exclusive access to a node). [8] However, we found many test cases (24.14% across projects) with 0 execution time as they took less than 0.0001 seconds to run. [9] Future research should consider evaluating TSM approaches on much larger industrial systems.

3.6 Conclusion

In this chapter, we proposed LTM, a scalable and black-box similarity-based test suite minimization (TSM) approach based on pre-trained language models. We investigated five alternative language models—CodeBERT, GraphCodeBERT, UniXcoder, StarEncoder, and CodeLlama—to extract test method embeddings and two similarity measures (i.e., Cosine Similarity and Euclidean Distance) for similarity calculation based on these embeddings. We employed an optimized version of the Genetic Algorithm (GA) by utilizing an efficient data structure for handling similarity values to improve fitness calculation and find optimal subsets of test suites. We assessed the performance of alternative LTM configurations on a large dataset consisting of 17 Java projects with a total of 835 versions collected from DEFECTS4J. We used the Fault Detection Rate (*FDR*), Minimization Time (*MT*), and Time Saving Rate (*TSR*) as evaluation metrics to assess the effectiveness and efficiency of LTM. We identified the best configuration of LTM and compared it to ATM, a recent black-box TSM approach that achieved better trade-off in terms of effectiveness and efficiency than the alternative SOTA TSM approach (FAST-R). Results indicate that the best LTM configuration (i.e., UniXcoder/Cosine) outperformed the two best configurations of ATM by achieving significantly higher *FDR* results (0.84 versus 0.81, on average) and more importantly, running much faster (0.82 min versus 4.06 min, on average) than ATM, in terms of both preparation time (up to two orders of magnitude faster) and search time (up to one order of magnitude faster), which is particularly important on much larger industrial

[8] We further collected the minimum and maximum execution times for each test method across the 10 runs. The results show that the difference in *TSR*, based on minimum and maximum execution times, ranges from 0.05% to 1.38% across all configurations, which indicates that the variation in test method execution times across runs is small and does not impact our conclusions regarding *TSR*.

[9] To ensure this did not impact our results, we investigated the effect of these test cases on the Time Saving Rate by assuming their execution times to be 0.0001—their maximum possible value—and recalculating the Time Saving Rate. We found that the average Time Saving Rate of all configurations slightly increased by 0.08% to 0.09% only, which is negligible and thus does not invalidate our conclusions.

systems and test suites, while achieving a saving of 41.72% in test execution time for a 50% test minimization budget.

Future work. We plan to assess the performance of LTM in practice using large industrial software systems and projects in other programming languages to further generalize our conclusions. The main challenge is the ability to automatically trace test failures to system faults.

3.7 Data Availability

The replication package of our experiments, including the data, code, results for other minimization budgets, and detailed *FDR*, *MT*, and *TSR* results of LTM and baseline approaches, is available on Zenodo [52].

Chapter 4

Conclusion

In this chapter, we summarize the contributions of this thesis, discuss its limitations, and identify future work directions.

4.1 Summary of Contributions

In summary, this thesis provides the following contributions:

1. ATM, a black-box test suite minimization technique based on tree-based test code similarity and evolutionary search. Our contributions related to ATM include:
   (a) ATM achieves a better trade-off between effectiveness (i.e., fault detection rate) and efficiency (i.e., minimization time) than the state-of-the-art (FAST-R) through finer-grained similarity analysis and a dedicated search algorithm.
   (b) An evaluation of the effectiveness and efficiency of ATM on a large dataset of 16 Java projects with 661 faulty versions using three budgets ranging from 25% to 75% of test suites.
2. LTM, a novel, scalable, and black-box similarity-based test suite minimization technique based on large language models. Contributions related to LTM are:
   (a) A black-box test suite minimization technique that attains a significantly higher fault detection rate and is much more scalable than the state-of-the-art (i.e., ATM) by minimizing test suites nearly five times faster on average.
   (b) An comprehensive comparison of five recent language models, varying in architecture, parameter size, input, and pre-training tasks.
   (c) An optimized search process using a sparse matrix for fitness calculation, which largely reduces the minimization time and memory usage.
   (d) A large-scale empirical evaluation of the effectiveness and efficiency of LTM and its comparison with ATM.

4.2 Limitations

The limitations of the results can be summarized as follows:

- **Scalability** Our results show that LTM runs five times faster than ATM on average, with higher gains for larger test suites and systems, thus achieving much higher scalability. However, LTM still presents limitations in scalability as the minimization time increases quadratically with the number of test cases. This can be improved by parallelizing the similarity and fitness calculation process, though it requires more resources (GPU and CPU cores).
- **Dataset** Both LTM and ATM are evaluated on the DEFECTS4J dataset, which contains test cases that are exclusively written in Java. The conclusions regarding LTM's performance may be specific to the characteristics of Java code and thus cannot be generalized to software systems in other languages. Moreover, the largest test suite size in DEFECTS4J is 7308, which is still relatively small compared to test suites in many real-world software systems. This limits our ability to generalize our conclusions regarding the scalability of LTM for much larger test suites in practice. We need to adapt and evaluate our work using much larger test sets in other programming languages to further generalize our conclusions.
- **Test code input length** For each project version, there are an average of 2.98% of the test cases that exceed the token length limit (512) of the language models, which can cause information loss and impact the performance of our work. Other language models with longer token length limits should be investigated, though they might impact the efficiency of test minimization.
- **Test code preprocessing** Our work LTM can be used with the test code without preprocessing and yields better results than using preprocessed test code as input. However, the test cases in many industrial systems contain duplicate and redundant information that can introduce noise to the similarity values. Moreover, preprocessing can also help in reducing the length of the test code inputs when they exceed the token length limit.

4.3 Future Work

Improve the scalability of TSM techniques. Though we have utilized language models, a highly optimized function for similarity calculation and an optimized search algorithm to accelerate the minimization process, there is still room for improvement in terms of the scalability of TSM. The similarity and fitness computation can be parallelized using multiple cores, thus significantly improving scalability. Although the parallelizing step requires additional computational resources (e.g., multiple CPU cores or GPUs), the cost can be offset when the test suite is extremely large, e.g., millions of test cases, because of the significant reduction in computation time. Moreover, the required memory is another

key factor that affects the scalability of TSM as it also increases rapidly with the test suite size, parallelizing the similarity calculation using GPUs largely reduces the memory required for each computation core, thus increasing memory usage efficiency.

Assessing the performance of our work on industrial systems. While DEFECTS4J is based on *real* faults that were extracted from open-source Java projects, thus enabling advanced and reproducible software testing research, it still has certain limitations that affect the generalizability of conclusions regarding the performance of our work. We plan to assess the performance of LTM in practice using large industrial software systems and projects in other programming languages to further generalize our conclusions. The key challenge is to automatically trace test failures to system faults, which is crucial for evaluation.

Investigating approaches to handle longer test code inputs. Our analysis indicates that some projects in our dataset have a higher percentage of test cases where the source code length exceeds the token length limits. This can impact the effectiveness of the similarity calculation and thus affect the fault detection capability of LTM. We plan to investigate the use of language models with longer token limits, though this comes at the cost of increased computational resources and longer running times. Moreover, we plan to investigate the use of certain preprocessing steps, such as normalizing variable identifiers, as they do not affect the data flow within the source code and thus can potentially reduce the test code length without impacting the effectiveness of similarity calculation.

References

[1] LTM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search – Replication Package. `https://doi.org/10.5281/zenodo.7455766`.

[2] Zohreh Aghababaeyan, Manel Abdellatif, Mahboubeh Dadkhah, and Lionel Briand. Deepgd: A multi-objective black-box test selection approach for deep neural networks. *ACM Transactions on Software Engineering and Methodology*, 2023.

[3] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2009.

[4] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[5] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology*, 114:137–154, 2019.

[6] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[7] Olivier Bachem, Mario Lucic, and Andreas Krause. Scalable k-means clustering via lightweight coresets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1119–1127, 2018.

[8] J. Blank and K. Deb. Pymoo: Multi-objective optimization in Python. *IEEE Access*, 8:89497–89509, 2020.

[9] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[10] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[11] Zhiyuan Chang, Mingyang Li, Junjie Wang, Qing Wang, and Shoubin Li. Putting them under microscope: a fine-grained approach for detecting redundant test cases in

natural language. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1161–1172, 2022.

[12] Tsong Yueh Chen and Man Fai Lau. Heuristics towards the optimization of the size of a test suite. *WIT Transactions on Information and Communication Technologies*, 14, 1970.

[13] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[14] Carmen Coviello, Simone Romano, Giuseppe Scanniello, Alessandro Marchetto, Giuliano Antoniol, and Anna Corazza. Clustering support for inadequate test suite reduction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–105. IEEE, 2018.

[15] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. Scalable approaches for test suite reduction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 419–429. IEEE, 2019.

[16] Heleno de S. Campos Junior, Marco Antônio P Araújo, José Maria N David, Regina Braga, Fernanda Campos, and Victor Ströele. Test case prioritization: a systematic review and mapping of the literature. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pages 34–43, 2017.

[17] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[19] Michel Marie Deza and Elena Deza. Encyclopedia of distances. In *Encyclopedia of distances*, pages 1–583. Springer, 2009.

[20] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[21] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

[22] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering*, 2022.

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[24] Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*, 2021.

[25] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[26] Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *international journal of Computer Applications*, 68(13):13–18, 2013.

[27] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[28] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[29] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Reducing the cost of model-based testing through test case diversity. In *Testing Software and Systems: 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings 22*, pages 63–78. Springer, 2010.

[30] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):1–42, 2013.

[31] Kim Herzig. Testing and continuous integration at scale: Limits, costs, and expectations. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, pages 38–38, 2018.

[32] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[33] Thorsten Joachims et al. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. In *ICML*, volume 97, pages 143–151. Citeseer, 1997.

[34] William B Johnson. Extensions of lipschitz mappings into a hilbert space. *Contemp. Math.*, 26:189–206, 1984.

[35] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126, 2021.

[36] Saif Ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines. *IEEE Access*, 6:11816–11841, 2018.

[37] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

[38] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International conference on machine learning*, pages 957–966. PMLR, 2015.

[39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[40] Yue Liu, Kang Wang, Wang Wei, Bofeng Zhang, and Hailin Zhong. User-session-based test cases optimization method based on agglutinate hierarchy clustering. In *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, pages 413–418. IEEE, 2011.

[41] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[42] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available at `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[43] Xiaofei Ma, Zhiguo Wang, Patrick Ng, Ramesh Nallapati, and Bing Xiang. Universal text representation from bert: An empirical study. *arXiv preprint arXiv:1910.07973*, 2019.

[44] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. A static approach to prioritizing junit test cases. *IEEE transactions on software engineering*, 38(6):1258–1275, 2012.

[45] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[46] Breno Miranda and Antonia Bertolino. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software*, 131:528–549, 2017.

[47] Antonio J Nebro, Juan J Durillo, Francisco Luna, Bernabé Dorronsoro, and Enrique Alba. Mocell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, 24(7):726–746, 2009.

[48] Raphael Noemmer and Roman Haas. An evaluation of test suite minimization techniques. In *International Conference on Software Quality*, pages 51–66. Springer, 2020.

[49] Robert E Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3-4):225–236, 1985.

[50] Rongqi Pan, Taher A. Ghaleb, and Lionel Briand. Atm: Black-box test case minimization based on test code similarity and evolutionary search. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2023.

[51] Rongqi Pan, Taher A Ghaleb, and Lionel Briand. LTM: Scalable and Black-box Similarity-based Test Suite Minimization based on Language Models. *arXiv preprint arXiv:2304.01397*, 2023.

[52] Rongqi Pan, Taher A. Ghaleb, and Lionel C. Briand. LTM: Scalable and Black-box Similarity-based Test Suite Minimization based on Language Models (replication package). `https://doi.org/10.5281/zenodo.13685828`, 2023.

[53] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. Fastlane: Test minimization for rapidly deployed large-scale online services. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE, 2019.

[54] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[55] Michel Raymond and François Rousset. An exact test for population differentiation. *Evolution*, pages 1280–1283, 1995.

[56] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[57] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*, 2023.

[58] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[59] Lucas Pereira da Silva and Patrícia Vilain. LCCSS: A similarity metric for identifying similar test code. In *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 91–100, 2020.

[60] ANRL Sirisha and Ashok Kumar Pradhan. Cosine similarity based directional comparison scheme for subcycle transmission line protection. *IEEE Transactions on Power Delivery*, 35(5):2159–2167, 2019.

[61] Peter D Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of artificial intelligence research*, 37:141–188, 2010.

[62] Gabriel Valiente. *Algorithms on trees and graphs*. Springer Science & Business Media, 2002.

[63] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. File-level vs. module-level regression test selection for .NET. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 848–853, 2017.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[65] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. Identifying similar test cases that are specified in natural language. *IEEE Transactions on Software Engineering*, 2022.

[66] MK Vijaymeena and K Kavitha. A survey on similarity measures in text mining. *Machine Learning and Applications: An International Journal*, 3(2):19–28, 2016.

[67] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software*, 103:370–391, 2015.

[68] Shuyin Xia, Zhongyang Xiong, Yueguo Luo, Guanghua Zhang, et al. Effectiveness of the euclidean distance in high dimensional spaces. *Optik*, 126(24):5614–5619, 2015.

[69] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[70] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[71] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 192–201. IEEE, 2013.

[72] Man Zhang, Shaukat Ali, and Tao Yue. Uncertainty-wise test case generation and minimization for cyber-physical systems. *Journal of Systems and Software*, 153:1–21, 2019.

[73] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report*, 103, 2001.