

ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search

Rongqi Pan
School of EECS
University of Ottawa
Ottawa, Canada
rpan099@uottawa.ca

Taher A. Ghaleb
School of EECS
University of Ottawa
Ottawa, Canada
tghaleb@uottawa.ca

Lionel Briand
School of EECS, University of Ottawa
Ottawa, Canada
SnT Centre, University of Luxembourg
Luxembourg
lbriand@uottawa.ca

Abstract—Executing large test suites is time and resource consuming, sometimes impossible, and such test suites typically contain many redundant test cases. Hence, test case (suite) minimization is used to remove redundant test cases that are unlikely to detect new faults. However, most test case minimization 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 was proposed, called FAST-R, relying solely on test case code for test case minimization, which appeared to be much more efficient than white-box techniques. However, it achieved a comparable low fault detection capability for Java projects, thus making its application challenging in practice. In this paper, we propose ATM (AST-based Test case Minimizer), a similarity-based, search-based test case minimization 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. ATM transforms test case code into Abstract Syntax Trees (AST) and relies on four tree-based similarity measures to apply evolutionary search, specifically genetic algorithms, to minimize test cases. We evaluated 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. 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), given that minimization is only occasionally applied when many new test cases are created (major releases). Results achieved for other budgets were consistent.

Index Terms—Test case minimization, Test suite reduction, Tree-based similarity, AST, Genetic algorithm, Black-box testing

I. INTRODUCTION

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 [1], if not infeasible. Such test suites are prone to similar and redundant test cases

This work was supported by a research grant from Huawei Technologies Canada Co., Ltd, as well as by the Mitacs Accelerate Program, the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC). The experiments conducted in this work were enabled in part by support provided by the Digital Research Alliance of Canada (<https://alliancecan.ca>).

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 [1], especially for large industrial systems, thus warranting systematic and automated strategies to eliminate redundant test cases, known as test case (suite) minimization.

Though many test case minimization techniques exist [2], 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 [3, 4]. One exception is the recent and novel work of Cruciani et al. [5], 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 comparable low fault detection capability for Java test cases. Unlike test case selection and prioritization, test case minimization is typically performed on an occasional basis [6], 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.

In this paper, we propose ATM (AST-based Test case Minimizer), a test case minimization 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 preprocesses 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 Genetic Algorithm (GA) and its multi-objective counterpart, Non-Dominated Sorting Genetic Algorithm II (NSGA-II), to minimize test cases 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. [5], 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 [7, 8], which has been shown to achieve better results than those at the class level [9]–[11]. We used the Fault Detection Rate (*FDR*) and execution time 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 case 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 case 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).

Overall, this paper makes the following contributions.

- A black-box, AST-similarity- and search-based test case minimization 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 test case minimization experiment on 16 projects with 661 versions comparing several configura-

tions 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 test case minimization experiment to date in the research literature.

The rest of this paper is organized as follows. Section II presents our proposed technique for test case minimization. Section III presents the experiment design, reports experimental results, and discusses their practical implications. Section IV discusses the validity threats to our results. Section V reviews and contrasts our technique with related work. Finally, Section VI concludes the paper and suggests future work.

II. ATM: BLACK-BOX TEST CASE MINIMIZATION

This section describes our black-box technique, called ATM, for test case minimization relying on tree-based test code similarity and evolutionary search. Figure 1 gives an overview of the main steps of ATM. We first describe how we pre-process the source code of test cases (Section II-A) and transform them into ASTs (Section II-B). Then, we describe the algorithms we employed for measuring the similarity between these ASTs (Section II-C). Finally, we describe the search-based algorithms we used to minimize test cases using similarity measures as fitness (Section II-D).

A. 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. [12], 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 methods¹, including their parameters.
- 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.

¹<https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html>

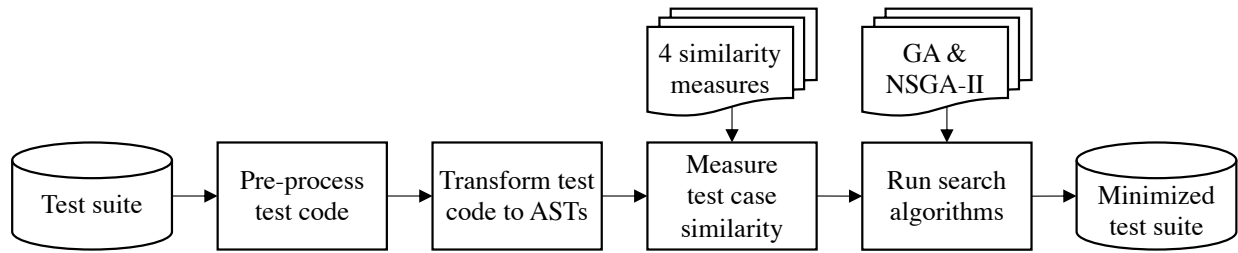


Fig. 1: An overview of the main steps of ATM to perform test case minimization

Listing 1 and 2 show a sample test case code before and after pre-processing, respectively.

```

1 /**
2  * Confirm that the equals method can distinguish
3  * all the required fields.
4  */
5 public void testEquals() {
6     DefaultTableXYDataset d1=new
7     DefaultTableXYDataset ();
8     DefaultTableXYDataset d2=new
9     DefaultTableXYDataset ();
10    assertTrue(d1.equals(d2));
11    assertTrue(d2.equals(d1));
12    d1.addSeries(createSeries1());
13    assertFalse(d1.equals(d2));
14    d2.addSeries(createSeries1());
15    assertTrue(d1.equals(d2));
16 }
  
```

Listing 1: Test case before pre-processing

```

1 public void test_case() {
2     DefaultTableXYDataset id_1=new
3     DefaultTableXYDataset ();
4     DefaultTableXYDataset id_2=new
5     DefaultTableXYDataset ();
6     id_1.addSeries(createSeries1());
7     id_2.addSeries(createSeries1());
8 }
  
```

Listing 2: Test case after pre-processing

B. 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 [13], 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 [14]. To do this, we used an AST parser, provided by the Eclipse JDT library², 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 important as it represents the structure of test case code. Therefore, ASTs are considered *labeled, ordered trees*.

C. Similarity Measures

We used different algorithms to perform tree-based similarity measurement 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 test case minimization 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 [15] for more details.

1) *Top-down similarity measure*: This measure is based on the top-down maximum common subtree isomorphism algorithm [15]. Tree isomorphism determines whether the nodes of one tree has 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-

²<https://www.eclipse.org/jdt>

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.

2) *Bottom-Up similarity measure*: This measure is based on the bottom-up maximum common subtree isomorphism algorithm [15]. 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 [15]. 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 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.

3) *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.

4) *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 [15], which is commonly used for tree comparison. To do this, a sequence of elementary edit operations are applied to one tree until the other tree is

obtained. Tree edit distance is not expected to be efficient when compared to previous similarity measures, especially for large ASTs, but is nevertheless an option.

5) *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.

$$Sim_{MaxCommonSubtree}(T_1, T_2) = \frac{2 \times |V_m|}{|V_1| + |V_2|} \quad (1)$$

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.

$$Sim_{TreeEditDistance}(T_1, T_2) = \frac{|V_1| + |V_2| - d}{|V_1| + |V_2|} \quad (2)$$

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.

6) *Similarity measurement implementation*: We used an open-source library, called **simpack**³ for implementing the algorithms for top-down and bottom-up maximum ordered 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 [15] (available in our replication package [16]). 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.

D. Search-based Test Case Minimization

Considering that test case minimization 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].

1) *Genetic Algorithm (GA)*: This is the most widely used search algorithm in search-based software testing, inspired by evolutionary theory. Applying GA requires to properly define (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

³<https://files.ifi.uzh.ch/ddis/oldweb/ddis/research/simpack>

test case minimization in our context, where test cases with higher similarity 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 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 [18] that keeps only test cases belonging to both parent test subsets, then proceed 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 inversion operator to randomly select a segment of an individual and reverse its order [18] to ensure fixed-size offspring. The parameters used for our GA are consistent with what is recommended in published guidelines⁴. 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.

2) *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 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 [19]. 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.

III. VALIDATION

This section reports on the experiments we conducted to evaluate the effectiveness and efficiency of ATM. As men-

tioned 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.

A. Research Questions

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

Test case minimization 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. 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. 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 cases?** 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 test case minimization 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 cases?** Test case minimization 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 cases. 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 minimization time (taken for running search algorithms).

RQ2: How does ATM compare to state-of-the-art black-box test case 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 test case minimization techniques [2], both white-box and black-box, were proposed to remove redundant test cases that are likely to detect the same faults. However,

⁴<https://www.obitko.com/tutorials/genetic-algorithms/recommendations.php>

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 V, 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 case minimization, a standard baseline, and (b) FAST-R, a set of novel black-box test case minimization 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 execution time.

B. 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.

1) *Minimization budgets*: For experimental purposes, the minimization budgets were set at 25%, 50%, and 75% of the test suites, for the reasons mentioned above. 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 [16].

2) *Baseline techniques*:

Random minimization. We used random minimization of test cases 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 [20]. 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 test case minimization techniques proposed by Cruciani et al. [5], 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 [21], and based on these vectors, test cases are then clustered using *k*-means++ [22] (FAST++) and constructed coresets [23] (FAST-CS). FAST-pw and FAST-all, however, use minhashing and locality-sensitive hashing [24] 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 execution 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-R⁵ provided by its authors. Though 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.

3) *Dataset*: We evaluated ATM compared to the baseline techniques on 16 Java projects collected from a public dataset, called DEFECTS4J⁶, 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 paper, 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

⁵<https://github.com/ICSE19-FAST-R/FAST-R>

⁶<https://github.com/rjust/Defects4J>

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 test case minimization 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 III-D. 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 test case minimization at the method level helps remove unnecessary test cases in a more precise manner [7, 8], which has been shown to achieve better results than those at the class level [9]–[11]. 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 II-C. 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.

4) Evaluation metrics:

Fault Detection Rate (FDR). Test case minimization aims to remove redundant test cases for a given budget while maintaining high *FDR*. Therefore, we used *FDR* to assess the effectiveness of ATM. *FDR* was calculated for each project as follows.

$$FDR = \frac{\sum_{i=1}^m f_i}{m} \quad (3)$$

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.

Fisher’s exact test. We used Fisher’s exact test [25], a non-parametric statistical hypothesis test, to assess how significant is the difference in proportions of detected faults between the alternative ATM configurations.

Odds ratio. We used the odds ratio [26] as an effect size measure of the magnitude of improvement of one ATM configuration over another. An odds ratio of 1 indicates no difference between two techniques, whereas an odds ratio of > 1 indicates a higher chance for one technique to perform better than the other.

Execution time. Execution 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 *minimization time*, taken to run search algorithms. Note that, when reporting execution time results, we did not consider the savings in execution time 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 execution time. For each ATM configuration, we computed the average execution time for each project version. We also computed the execution times taken by baseline techniques.

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

$$Fitness = \frac{\sum_{i,t_i \in M_n} \text{Max}_{j,t_j \in M_n, i \neq j} \text{Sim}(t_i, t_j)^2}{n} \quad (4)$$

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

C. 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 [16]).

1) *RQ1 results*: Table I reports the *FDR* and total execution time (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 I shows that all ATM configurations achieved high *FDR* results (mean ≥ 0.74 and median ≥ 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 [16]. 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. Execution time results in Table I combines both preparation time and minimization time for each project version (see detailed results in our replication package [16]). We observe that the average execution time 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 test case minimization, where redundant test cases are removed on an occasional basis when many new test cases are created for major releases, such execution times are acceptable in practice, as further discussed below. However, the execution time 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 execution time, 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 execution 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 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.

Minimization Time. We found that the time ATM took for minimizing test cases 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 minimization time, since it converged faster, than combined similarity using GA. However, when considering the execution time for the

TABLE I: Descriptive statistics of *FDR* and total execution time (in minutes) of ATM across project versions for the 50% minimization budget. The highest *FDR* and shortest execution time 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	
	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>
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

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%).

2) *RQ2 results:* Table II compares, in terms of *FDR* and total execution time 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.

Execution time results. We observe that all FAST-R alternatives ran much faster than ATM, in terms of both preparation and minimization, with an average total execution time 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 Test case minimization is

⁷As 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.

typically performed on an occasional basis [6], 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.

D. Discussion

Effective test case 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 case 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 simi-

TABLE II: Descriptive statistics of *FDR* and total execution time (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 execution time are highlighted in bold

Statistic \ Technique	ATM GA/Combined		FAST++		FAST-CS		FAST-pw		FAST-all		Random minimization	
	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>	<i>FDR</i>	<i>Time</i>
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

larity, it took relatively less time to minimize test cases as it converges faster on all projects. Still, when accounting for the total execution 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 test case minimization 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 test case minimization experiment is the largest to date, industrial systems can be much larger than the ones in our dataset. Overall, we observed that the execution 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 [27, 28]. Other search algorithms [19] should also be investigated in the future to identify better trade-offs between efficiency and effectiveness.

Application of test case minimization in practice. In practice, when there is a major release with many new test cases created and a testing budget is set, similarity scores 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. The resulting minimized test suite is later used for regression testing in the subsequent code versions.

IV. 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 on the same finer-grained data. We observe that FAST-R’s performance was consistent with what was originally reported [5], 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, 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.

V. RELATED WORK

Test case minimization is an NP-hard problem, hence there are many techniques that have been proposed [1, 2] 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 case minimization. Miranda et al. [29] used greedy heuristics [30] to perform test case minimization 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. [6] also used greedy heuristics to perform test case minimization 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 case minimization. Cruciani et al. [5] recently proposed a novel technique, called FAST-R, to perform test case minimization by relying solely on test case code (black-box), which is converted into vectors using a term frequency model [21]. 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. [31] and Viggiano et al. [32] proposed clustering-based test cases minimization 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 case minimization. Hemmati et al. [33] used a search algorithm that relies on test case similarity calculated using model-based features to perform test case minimization. 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. [34] and Wang et al. [35] proposed model-based test case minimization techniques relying on multi-objective search algorithms, such as NSGA-II, MOCell [36], SPEA2 [37]. 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) execution time. 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 test case minimization.

VI. CONCLUSION

In this paper, we proposed ATM, a black-box test case (suite) minimization 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 test case minimization 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 execution time 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 test case minimization 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%).

Future work. We aim to extend ATM in the future to use additional similarity measures, such as Normalized Compression Distance (NCD) [38], and other code representation techniques using language models, such as CodeBERT [39] and TreeBERT [40]. We also aim to expand our evaluation to consider projects using other programming languages and larger industrial systems.

VII. DATA AVAILABILITY

The replication package of our experiments, including the data, code, results for other minimization budgets, and detailed *FDR* and execution time results of ATM and baseline techniques, is available on Zenodo [16].

REFERENCES

- [1] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] Raphael Noemmer and Roman Haas. An evaluation of test suite minimization techniques. In *International Conference on Software Quality*, pages 51–66. Springer, 2020.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] Heleno de S. Campos Junior, Marco Antônio P Araújo, José Maria N David, Regina Braga, Fernanda Campos, and Victor Ströle. 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.
- [12] 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.
- [13] 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.
- [14] Robert E Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3-4):225–236, 1985.
- [15] Gabriel Valiente. *Algorithms on trees and graphs*. Springer Science & Business Media, 2002.
- [16] ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search – Replication Package. <https://doi.org/10.5281/zenodo.7455766>.
- [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] J. Blank and K. Deb. Pymoo: Multi-objective optimization in Python. *IEEE Access*, 8:89497–89509, 2020.
- [19] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available at <http://cs.gmu.edu/~sean/book/metaheuristics/>.
- [20] 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.
- [21] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.
- [22] 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.
- [23] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.
- [24] Michel Raymond and François Rousset. An exact test for population differentiation. *Evolution*, pages 1280–1283, 1995.
- [25] 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.
- [26] Erick Cantú-Paz et al. A survey of parallel genetic algorithms. *Calculateurs parallèles, réseaux et systèmes repartis*, 10(2):141–171, 1998.
- [27] 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.
- [28] Breno Miranda and Antonia Bertolino. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software*, 131:528–549, 2017.
- [29] 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.
- [30] 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.
- [31] Markos Viggiano, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. Identifying similar test cases that are specified in natural language. *IEEE Transactions on Software Engineering*, 2022.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] Antonio J Nebro, Juan J Durillo, Francisco Luna, Bernabé Dorronsoro, and Enrique Alba. Mocell: A cellular genetic algorithm for multi-objective optimization. *International Journal of Intelligent Systems*, 24(7):726–746, 2009.
- [36] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report*, 103, 2001.
- [37] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 223–233. IEEE, 2016.
- [38] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- [39] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. TreeBERT: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR, 2021.