ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development

Maral Azizi
Departemant of Computer Sceince and Engineering
University of North Texas
maralazizi@my.unt.edu

Hyunsook Do
Departemant of Computer Sceince and Engineering
University of North Texas
hyunsook.do@unt.edu

***Abstract*—Regression test selection offers cost savings by selecting a subset of existing tests when testers validate the modified version of the application. The majority of test selection approaches utilize static or dynamic analyses to decide which test cases should be selected, and these analyses are often very time consuming. In this paper, we propose a novel language-independent Regression TEst SelecTion (ReTEST) technique that facilitates a lightweight analysis by using information retrieval. ReTEST uses fault history, test case diversity, and program change history information to select test cases that should be rerun. Our empirical evaluation with four open source programs shows that our approach can be effective and efficient by selecting a far smaller subset of tests compared to the existing techniques.**

I. Introduction

Regression testing is one of the important maintenance activities that can control the quality and reliability of modified software, but it can also be very expensive. The cost of regression testing can account for one half of the software maintenance costs [48]. To date, numerous regression testing techniques (e.g., test selection, test prioritization, and test reduction) have been introduced to reduce the cost of regression testing [11], [20], [38], [48]. The majority of regression testing techniques have relied on code coverage information and many empirical studies have shown that these techniques can improve the effectiveness of regression testing [8], [15], [22], [34], [48], [49]. However, code coverage information is usually unavailable before test cases are executed, and often collecting coverage information requires a considerable amount of time, in particular for large industrial applications. Moreover, as software systems evolve over time, such information can become incomplete and imprecise due to code changes and addition/deletion of tests, and thus the coverage collection task should be repeated over system lifetimes [35], [43].

Fault history is another type of information that has been actively utilized in developing regression testing techniques [18], [27], [38], [46], [50]. In these techniques, testers prioritize tests based on the failure history of tests. The underlying hypothesis of these techniques is that tests that detected faults in the past have a higher chance of catching faults in future versions. However, these techniques suffer from the same limitations as code coverage-based techniques do. A test suite often changes as software evolves, so the collected failure history information from previous versions might need to be updated for new versions. Further, for the newly added tests, no failure history is available. Therefore, such test cases would be ignored during the test selection and execution phase while they might have a higher chance to reveal faults than other tests.

These limitations can be more problematic in modern software development because the speed of software delivery has become one of the important success factors for modern software applications. Further, agile methodologies emphasize the importance of fast software deployment [13] and companies providing web based applications have taken this to extremes. For example, Amazon deploys software every 11.6 seconds on average [29]. To meet this demand, efficient testing approaches that can be aligned with today's incremental and iterative software delivery practices should be considered.

To address these limitations, in this paper, we propose an automated lightweight Regression TEst SelecTion (ReTEST) approach that utilizes information retrieval techniques. Information retrieval (IR) techniques have been applied to solve various software engineering problems [14], [31], [37], [43], [44]. These techniques rely on relevant textual information extraction from source code and other software artifacts (e.g., code commits, requirement documents, etc) and provide fast and lightweight data extraction from a large amount of information. Our technique uses multiple test quality metrics to improve the effectiveness and efficiency of regression testing techniques, focusing on regression test selection.

To implement ReTEST, we used program changes, test suite source files, and fault history information. Using these types of information, ReTEST identifies a list of important test cases that are highly likely to detect regression defects. The primary inputs for ReTEST are: 1) term similarity to the modified portion of the program, 2) test failure history, and 3) test diversity. Typically, regression faults occur due to code changes, therefore code change information is one of the important metrics to consider when locating regression faults [24], [46], [48]. We also used test failure information because previous studies have shown that if a test detects a fault in the past, then that test is likely to fail again because it exercises a part of the program that used to be faulty and there is a high chance that this part will be faulty again [21], [30], [38], [39]. We also used test dissimilarity because empirical evidence indicates that diverse tests can produce high fault detection rates [25], [26], [33], [35].

To evaluate ReTEST, we performed an empirical study

using four open source applications written in two different programming languages. The experimental results show that our approach outperformed the existing techniques for the majority of applications under test. The results also show that ReTEST can be effective in practice by selecting a far smaller subset of test cases compared to the safe technique. Thus, ReTEST provides an effective alternative approach that addresses the test selection problem without requiring dynamic coverage information or static analysis. Furthermore, unlike traditional techniques, ReTEST is program language independent and can be applied to applications written in different languages. Here are our main contributions:

- We introduce a new cost effective regression test selection technique, which is language independent, fast, scalable, and lightweight. It facilitates a graph search technique to address the limitations of traditional regression testing problems and eliminates code coverage profiling.
- Our approach provides an efficient search algorithm to select the best candidate test cases through the generated graph database using the defined test quality metrics.
- We also empirically evaluated the proposed approach by comparing with three test selection techniques, which are commonly used for regression testing.

The rest of the paper is structured as follows. Section II presents the ReTEST approach. Sections III, IV and VI present our study, including design, data analysis, and threats to validity. Section V discusses the results including practical implications. Section VII presents related work, and Section VIII discusses conclusions and future work.

II. Approach

The goal of our approach is to improve the effectiveness as well as the efficiency of test selection. ReTEST addresses two main limitations of the traditional test case selection techniques: code coverage profiling overhead, and a problem with newly added tests. To achieve our goal, we propose an automated test selection technique that utilizes three sources of information (program change information, test suite source code, and test failure history). Our proposed system, ReTEST, is built based on a graph database in which nodes of the graph are test cases and the edges between nodes are the relationships among test cases. Figure 1 presents an overview of the proposed technique. First, ReTEST uses test case source files to create a graph database. To build the graph database, ReTEST tokenizes the tests and then calculates the diversity among test cases (the details are explained in Sections II-A1 and II-A2). Using the code change information between two consecutive program versions, we build queries to identify a set of test cases that likely exercise the modified portion of a program, which can increase the chances of fault detection. In the following subsections, we describe our proposed approach in detail including the test case graph database and a traversal algorithm with a walk-through example.

A. Construction of Test Case Graph

In this section, we explain how a graph database schema is defined. The graph network that we want to construct has

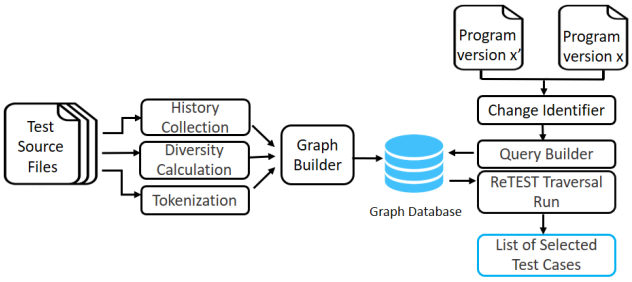


Fig. 1: An Overview of ReTEST Framework

three elements: property keys, vertex label, and edge label. The property keys are the types of properties that can be associated with vertices and edges. To define the property keys, we need three items: 1) unique key is the name of the property, which has to be unique; 2) data type is the data type of values; the property can have cardinality, which means that some properties can have multiple values; and 3) values are the content of the properties.

1) Network Vertices: Every vertex in our network needs a unique label, which describes the name of a vertex. Because the vertices in our network represent test cases, we define their label with 'T' plus a unique identifier value (T_i for test i). The next element we need to define for our network vertices is properties. Each vertex has three properties as follows:

1) Test source: includes the test case source code.
2) Fault history: is a Boolean value that represents whether a test case has been failed before or not.
3) Corresponding query: represents the query that is associated with a test.

2) Network Edges: After creating the network's vertices, we need to build the edges among them. In a graph, an edge determines the relationship between two vertices. Therefore, to create the links among vertices, we need to define a function that can measure the diversity among test cases. For example, assume that we have two test cases, 1 and 2. If test case 1 includes method calls A and B and test case 2 includes method calls B and D, then method B is a common entity between the two test cases, thus we link these two test cases. The diversity between two test cases is a distance function that measures their dissimilarity. The underlying hypothesis of measuring the test diversity is that the more diverse tests are, the higher chance of covering different portions of code. Therefore, by exercising different portions of code, chances of bug detection would be higher [33], [35]. So far, several distance functions that can be applied to regression testing techniques have been investigated by researchers, such as Hamming distance, Leveneshtein, Jaccard Index, etc [10], [16], [25], [26], [35], [44], [45]. In this work, we measured the similarity between tests using Jaccard Index because previous studies have shown that this function outperformed other techniques in terms of fault detection ability [25].

The Jaccard Index is a statistical formula applied to compare the similarity of sample sets as shown below. In this formula, the similarity between two sample sets A and B can be measured by the size of the intersection divided by the size of the

union of the sample sets ($Jaccard\ Index(A, B) = \frac{|A \cap B|}{|A \cup B|}$).

Figure 2 shows a sample of the test case network. The orange boxes in this figure represent the vertices properties with their values, and the label on the edges represents the dissimilarity value between test cases measured using the Jaccard Index. As shown in Figure 2, both tests T2 and T3 call class DateTime, therefore, these two tests are connected.

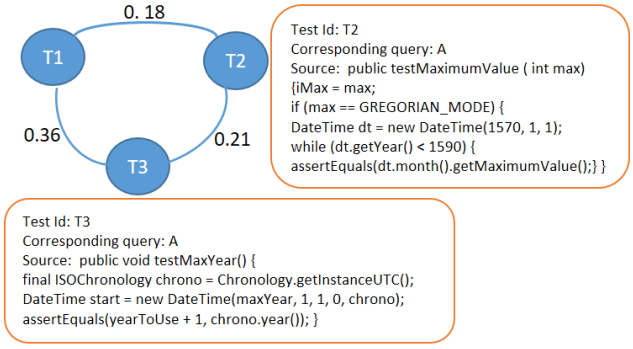


Fig. 2: An Example of a Network of Test Cases

B. Query Construction

Building a query is a critical problem in information retrieval [32]. Studies show that different types of the query construction may result in different outcomes [36]. To simplify the information extraction task in this work, we define queries as the program differences between two versions at the line level. In order to create the queries, we collect the code changes between two versions by applying a diff tool [1]. Before we apply the diff tool, we remove comments, spaces, and blank lines from each file. We also perform term filtering (i.e., deleting stop-words, such as articles and prepositions) and stemming that converts each word to its root form (e.g., converting a plural form to a singular form).

C. Search Query Algorithm

After building the network and queries, we need to determine the goal of query search in the network. To do this, we define our goals as follows: The first goal is to find test cases that have a high similarity to a query. The second goal is to identify a set of test cases that have been failed before. The third goal is to select the most diverse test cases among the collected test cases. The underlying hypothesis of these goals is that test cases that have higher cosine similarities to the queries are more likely to exercise the changed part of the program. Cosine similarity [9] is a measure of the relative similarity between two vectors, in our case, between a test case and a query ($Cosine\ (T_c, Q) = \frac{T_c.Q}{|T_c|.|Q|}$, where T_c is a test case and Q is a query).

For example, the vectors for T2 and T3 (Figure 2) are {max, imax, gregorianmode, dt, datetime, getyear, month, getmaximumvalue} and {isochronology, chrono, chronology, getinstanceutc, datetime, start, maxyear, yeartouse, year}, respectively. From the query (Figure 4), "month" and "imax" are found in T2 vector and "yeartouse" in T3 vector. The cosine similarity between tests and the query is 0.25 for T2, and 0.12 for T3 (The details about cosine similarity calculation can be found from various sources [9], [43]).

Further, previous studies have shown that if a test detects a fault in the past, that test is likely to fail again because it exercises a portion of the program that used to be faulty [12], [18], [50].Also, by selecting more diverse tests, the chance of exercising different portions of the program would be higher. Thus, the chance of bug detection would be higher [26].

Algorithm 1 shows the ReTEST traversal algorithm. The inputs of the algorithm are queries and test cases (lines 2 and 3) and the output is a set of selected tests that can satisfy the selection criteria (line 5). The S indicates the cosine similarity value of a test to a query and can be assigned by testers based on their goals (e.g., higher similarity, less number of test cases, etc.) (line 7) and the value N determines the percentage of the selected tests (line 8). The algorithm begins by receiving a set of queries and a set of test cases.

First, it selects test cases whose cosine similarity values are not smaller than the minimum assigned cosine similarity value to the query and adds them to the recommendation set (lines 14-21). The minimum cosine value is determined by testers. Note that in our tool, after extracting the queries, testers are informed about the range of cosine values. For example, if testers want to select test cases that exercise more modified portions of the program, they can assign a high value to the minimum cosine value (S). Then, based on the testing budget, testers decide the percentage (N) of tests to be selected from the recommendation set (line 22).

After building the recommendation set, the algorithm checks

Algorithm 1 ReTEST Traversal

```
1: Inputs:
2: Test[]    The set of all test cases from object of analysis.
3: Query[]   The set of program changes between two versions.
4: Output:
5: SelectionSet[]   The set of selected test cases for execution.
6: Declare:
7: S   The minimum similarity score of test cases to be selected for query matching.
8: N   The percentage of the tests to be selected for each query.
9: cap   The number of tests to be selected.
10: src   The source test to be selected first.
11: nn   The nearest test in the RecomSet with min dist of src.
12: RecomSet[]   The set of the tests that are similar to a specific query.
13: procedure TEST SELECTION(Test [], Query [])
14:     for each query q ∈ Query[] do
15:         for each test t ∈ Test[] do
16:             CalculateCosineSimilarity(t, q);
17:             if CosineSimilarity(t, q) >= S & t ∉ RecomSet[] then
18:                 RecomSet.Add(t);
19:             end if
20:         end for
21:     end for
22:     cap ← N * (RecomSet[].Size());
23:     for each t ∈ RecomSet[] do
24:         if t.hasFailed() & SelectionSet.Size() < cap then
25:             SelectionSet.Add(t);
26:             RecomSet.Remove(t);
27:         end if
28:     end for
29:     src ← SelectionSet[0];
30:     while SelectionSet.Size() < cap do
31:         nn = Min(JaccardDist(src, RecomSet[]));
32:         SelectionSet.Add(nn);
33:         src = nn;
34:         RecomSet.Remove(nn);
35:     end while
36:     return SelectionSet[];
37: end procedure
```

whether any of test cases in the recommendation set has failure history. If any of test cases has failed before, it is added to the selection set (lines 23-28). For the rest of test cases in $RecomSet$, ReTEST selects the first test from the selection set as a source test (line 29), it then calculates the distances of the source test from the rest of the tests in $RecomSet$, selects the test (nn) with the minimum distance (lowest similarity) of other tests in $RecomSet$ (this step is intended for selecting diverse test cases), adds the test (nn) to the selection set, remove the test from $RecomSet$ to avoid reexamining it, and updates the source test with nn (lines 30-35). This process is repeated until the size of $SelectionSet$ reaches cap (line 30). The size of the $SelectionSet$, cap, can be defined by testers based on their available time and resources.

D. An Example of Graph Generation and Traversal Procedure

Figure 4 shows an example of the ReTEST network layout. The top section of this figure shows a sample query that was extracted from program changes. This query was constructed from a sample code change (lines 115-121) in Figure 3. The query contains five key words ({"yeartouse", "month", "imax", "math", "signum"}) that we stemmed each word to its root form (e.g months to month, iMax to imax). The two common words if and else and symbols are also filtered. The middle section shows an overview of the graph database and the bottom section shows the results of the query retrieval. "Rank" indicates the rank of test cases based on their cosine similarity to the query. "Similarity Score" is the cosine similarity score between test cases and a query. "Failure" indicates whether the test case has been failed previously or not.

9 src/test/java/org/joda/time/chrono/BasicMonthOfYearDateTimeFiled.java

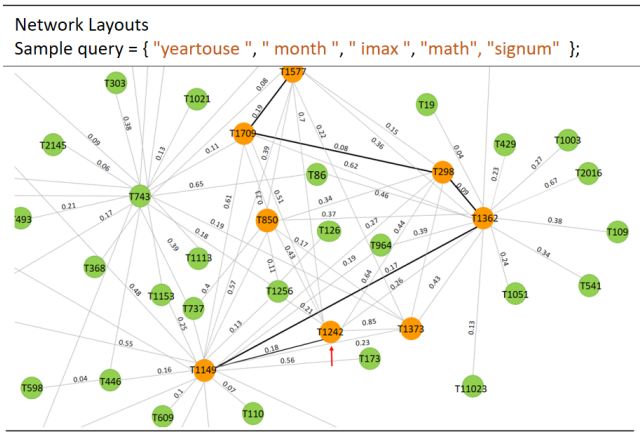
```
@@ -112,8 +112,13 @@ public long add(long instant, int months) {
        // Initially, monthToUse is zero-based
        int monthToUse = thisMonth - 1 + months;
        if (thisMonth > 0  && monthToUse < 0 ) {
-           yearToUse++;
-           months -= iMax;
+           if (Math.signum(months + iMax) == Math.signum(months)) {
+               yearToUse--;
+               months += iMax;
+           } else {
+               yearToUse++;
+               months -= iMax;
+           }
            monthToUse = thisMonth -  1  + months;
        }
        if (monthToUse >= 0 ) {
```

Fig. 3: An Example of Code Commit

The orange vertices represent tests that have high similarities with the query (i.e., T2 and T3 from Figure 2 share common words: yearToUse, month, and iMax, with the query in Figure 4). Suppose that the corresponding tests to the sample query are $\{T_{1149}, T_{1362}, ..., T_{1373}\}$ (as shown in the bottom section in Figure 4). Initially, ReTEST checks the testing budget. The testing budget is the assigned value of cap. For example, in this sample query, there are 8 test cases that are relevant to that query. If we set N = 0.75, then the testing budget is 6 (8 * 0.75), which means 75% of tests can be exercised due to the limited budget. ReTEST then sorts test cases based on their cosine similarity, and then it checks the test failure history. Among these eight test cases, T1242 has failed before, thus this test is added to the selected tests set.

Next, ReTEST has to find five additional test cases among seven candidates. Because there is no other failed test, ReTEST starts searching from test cases with the minimum similarity by calculating the distance of other candidate tests to this node, and selects tests with a minimum distance (the minimum distance indicates the minimum similarity between two tests). Note that, for a given query, there is a direct edge among all selected tests in $RecommSet$ because these tests share common terms between themselves and the query. Therefore, the minimum distance is always greater than zero.

As shown in Figure 4, among connected nodes to T1242, T1149 has the shortest distances (0.18), therefore, ReTEST selects this node as the next candidate. This process will be repeated until it reaches the budget limit. Therefore, the final output of ReTEST for this sample query would be $\{T_{1242}, T_{1149}, T_{1362}, T_{298}, T_{1709}, T_{1577}\}$.



Rank	1	2	3	4	5	6	7	8
Test Id	T1149	T1362	T298	T1577	T1709	T1242	T850	T1373
Similarity Score	0.3513	0.32751	0.3251	0.31687	0.2839	0.27502	0.2653	0.25075
Failure	0	0	0	0	0	1	0	0

Fig. 4: ReTEST Sample Query Extraction

III. Empirical Study

To assess our proposed technique, we performed an empirical study considering the following research questions:

- RQ1 How does ReTEST perform compared to the safe selection technique in terms of cost savings and fault detection?
- RQ2 How does ReTEST perform compared to diversity- and code coverage-based techniques in terms of fault detection?

A. Objects of Analysis

This study used four open source programs. Table I lists the applications and their associated data: "Version pair" (the program versions that we used for the experiment), "LOC" (the number of lines of code), "TstMethod" (the number of test cases in the method level), "TstClass" (the number of test cases in the class level), "NewTst" (the number of newly added test cases), "Faults" (the number of faults), "Queries"

(the number of queries that we built from the program change files), "SrcToken" (the number of tokens of source code), and "TstToken" (the number tokens of test files).

nopCommerce is an open source e-commerce shopping cart web application built on the ASP.Net platform [2]. *Umbraco* is a large-scale open source content management system written in $C\#$ language [3]. Two other applications were obtained from the defects4j database [4]: *JFreeChart* and *Joda-Time* (both are written in Java). All the faults for *JFreeChart* and *Joda-Time* are real and reproducible. The faults used in *nopCommerce* and *Umbraco* have been reported by users.[1] Also, all applications have multiple consecutive versions and test cases. In total, 27 versions of four programs were used to create the data required for evaluating the proposed technique.

B. Variables and Measures

1) Independent Variables: The independent variable in this study is regression test selection technique. We considered four test case selection techniques, which we classified into: three control techniques and one heuristic technique. Below we summarized these techniques. For our heuristic technique (ReTEST), we used the approach explained in Section II, thus we only explain the control techniques.

1) Safe Technique ($Safe$): selects all modification traversing tests, which guarantees 100% of fault detection.
2) Code Coverage (Cov): selects tests based on the total number of modified statements that they cover. If multiple tests cover the same number of statements, they are selected randomly.
3) Test Dissimilarity (Div): selects tests based on their dissimilarity in terms of the modified code exercised by them. Dissimilarity was calculated using Jaccard Index.

We consider one control technique for RQ1 ($Safe$), and two control techniques (Cov, and Div) for RQ2.

2) Dependent Variable and Measures: For RQ1, the dependent variables are fault detection ratio (FDR) and the number of selected test cases. For RQ2, the dependent variable is FDR ((detected-faults/total-faults)*100%).

C. Data Collection and Experimental Setup

As described earlier, ReTEST does not require any dynamic or static code coverage information, and it uses the term similarity among code commits and test cases. In addition to the term similarity, ReTEST uses two other test quality metrics to rank test cases. The first metric is fault history. To collect fault history information, we ran all test cases against faulty versions of each program to monitor each test's behavior, and then we store the results into the graph database. The second metric that we used is test diversity. To measure the dissimilarity among test cases, we followed the technique we explained in Section II-A2. Moreover, to compare the effectiveness of term similarity (S) against other test quality metrics, we examined ReTEST performance by assigning different values to the term similarity. Higher S values indicate that the focus of ReTEST is more on the term similarity than the test diversity, and vise versa. For example, when we set S = 0.4 (which is close to a maximum similarity value), only few tests would satisfy this condition and because ReTEST has to select a certain amount of test cases, it selects 100% of the test cases that can satisfy this condition. To collect code coverage information, we used the Visual Studio Test Analysis plugin for nopCommerce and Umbraco. For other two Java applications, we used the JaCoCo plugin on netbeans IDE [7]. We also collected test execution time using a PC with CPU Core i7, memory 16 GB, and Windows 10.

To compare ReTEST with the safe selection technique, we implemented the safe technique by following DejaVu approach [42]. DejaVu identifies all changes among versions and selects test cases that are affected by changes. For the code coverage-based technique, we built a greedy algorithm that selects tests that exercise the modified portion of code and prioritizes them based on their total code coverage. For the diversity-based technique, we measured the Jaccard Index among the set of test cases that exercise the changed portion of code, then we prioritized them based on their distance from each other. Because these two algorithms (code coverage and diversity) require random tie-breaking for tests with the same scores, we executed 30 independent runs of them per version for each application, and averaged the results.

IV. Data and Analysis

A. RQ1 Analysis

To answer RQ1, we compared the number of selected test cases, and Fault Detection Ratio (FDR) of ReTEST with those of the safe selection technique.

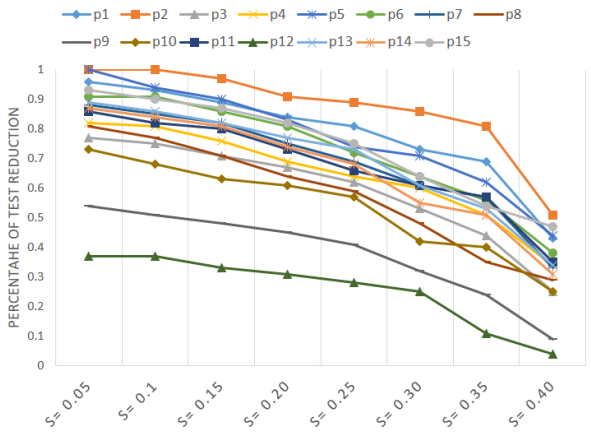


Fig. 5: Test Reduction Trends for Different S Values

Depending on the resource availability and budget, ReTEST can control the number of tests to be selected. However, ReTEST's outcome is affected by the similarity value that defines the initial set of tests, which is used for selecting the final set of tests considering the testing budget. Theoretically, the similarity value ranges between 0 and 1. By setting the similarity value close to 0, we have to consider almost all test cases as an initial set of tests. Conversely, by setting the similarity value close to 1, almost no tests will be considered as

[1]The reported faults for *nopCommerce* are available in the *GitHub* repository [5], and the bug history for *Umbraco* is accessible through their website [6].

TABLE I: Application Properties

No	Object	Version pair	LOC	# TstMethod	# TstClass	# NewTst	# Faults	# Queries	# SrcToken	# TstToken
P_1	nopCommerce	2.00 - 2.10	133,862	523	126	+5	28	65	17,803	2,768
P_2	nopCommerce	2.20 - 2.30	141,232	585	136	+3	32	118	18,052	2,893
P_3	Umbraco-CMS	7.7.0 - 7.7.2	312,511	4,129	628	+5	41	97	26,700	14,552
P_4	Umbraco-CMS	7.7.4 - 7.7.5	312,886	4,130	628	+2	28	131	26,811	14,546
P_5	Joda-Time	0.9 - 0.95	17,818	923	80	-70	10	143	3,328	2,767
P_6	Joda-Time	0.98 - 0.99	20,317	3,181	218	+242	10	128	4,905	3,765
P_7	Joda-Time	1.1 - 1.2	22,149	3,360	237	-188	21	130	5,290	4,368
P_8	Joda-Time	1.2 - 1.3	24,001	4,256	254	+896	21	115	5,380	4,622
P_9	Joda-Time	1.3 - 1.4	25,292	4,511	263	+225	12	54	5,496	4,812
P_{10}	Joda-Time	1.4 - 1.5	25,795	4,701	266	+10	18	112	5,610	5,106
P_{11}	JFreeChart	1.0.0 - 1.0.1	95,669	1,512	225	+11	12	98	14,219	6,039
P_{12}	JFreeChart	1.0.2 - 1.0.3	101,713	2,151	231	+571	10	6	11,855	4,750
P_{13}	JFreeChart	1.0.4 - 1.0.5	74,434	2,360	351	+83	13	101	12,057	5,102
P_{14}	JFreeChart	1.0.6 - 1.0.7	76,065	2,680	366	+196	8	85	12,662	5,522
P_{15}	JFreeChart	1.0.8 - 1.0.9	81,541	2,738	393	+35	11	186	12,738	5,636

an initial set of tests. Thus, finding a reasonable and practical similarity value is an important step for our approach.

Figure 5 shows the relationship between the similarity value (S) and the size of tests for the applications used in our study. As the figure shows, as S increases, the number of selected tests decreases. For the majority of the applications, with S = 0.05, more than 80% of test cases are selected as an initial set. Reversely, with S = 0.4, only 32% of tests are selected as an initial set on average. For some cases, the selection rate is close to 0. From these trends, we observe obvious tradeoffs between S value and the number of test cases: if we have enough time to run tests, then we can assign a low value to S; otherwise, we assign a high value to S. These tradeoffs provide a great flexibility when we apply regression testing techniques. Thus, in RQ1 investigation, we considered five different parameter settings (five experiments in Table II) to explore how our results vary on these settings and how ReTEST performs compared to the safe technique under various circumstances. Because S values in our experiment range between 0.038 to 0.42 (0.22 on average), we selected S values that are close to minimum and maximum values including three values between them. Different similarity values produce different numbers of initial tests. To control these numbers, we applied the testing budget when we obtained the initial set of tests. For example, we set the testing budget N to 30% in Experiment 1 in Table II, which means Experiment 1 considers 30% of tests that have 0.1 similarity because the similarity value with 0.1 would select almost all tests. But, for Experiment 5, we consider 100% of tests that have 0.35 similarity.

Table II presents the results from ReTEST and the safe selection technique. The table shows the number of selected tests, the percentage of saved tests, and the fault detection rate. Each subtable in Table II shows the results obtained by assigning different parameter values to ReTEST. S indicates the similarity score and N indicates the percentage of tests selected per each query. For instance, parameter setting in the first subtable is S = 0.1 and N = 30%. These values mean that for a sample query "a", ReTEST selects 30% of test cases with a similarity score that is greater than or equal to 0.1.

The first column lists the applications under test. The second and third columns present the number of selected tests for ReTEST and the safe technique, respectively. The fourth and fifth columns show the percentage of test reduction for ReTEST and the safe technique, respectively. The sixth column shows the improvement ratio of ReTEST over the safe technique. The seventh and eight columns present the number of missed faults and FDR for ReTEST, respectively. The safe technique guarantees to produce 100% of FDR.

Our quantitative results from Table II show that ReTEST was able to reduce the number of selected tests significantly. For instance, in Experiment 1, ReTEST was able to reduce the number of selected tests by 52% on average (ranging from 19.14% to 72.88%) compared to the safe technique. The results in Table II indicate that when we set S = 0.1 and N = 30%, the number of selected tests by ReTEST decreased to 53.5%, 59.5%, 56% and 65% on average for nopCommerce, Umbraco, JfreeChart and Joda-Time respectively. Also the maximum number of missed faults for each program is relatively high. Therefore, we can conclude that assigning a low similarity score to ReTEST reduces the effectiveness in terms of fault detection rate.

From Experiment 2, when we set S = 0.15 and N = 40%, the average number of selected tests for nopCommerce, JfreeChart, and Joda-Time increased by 13%, 14%, and 22% while it decreased to 18% for Umbraco compared to Experiment 1. Also, the number of missed faults decreased and the FDR values slightly increased. Overall, five out of the fifteen examined projects reached the maximum FDR. From this experiment, we can say that a minor increase in the similarity score would increase the percentage of fault detection.

In Experiment 3, when we set S = 0.20 and N = 50%, the average number of selected tests for nopCommerce, Umbraco, and JfreeChart increased by 5%, 21%, 0.17% and decreased by 4% for Joda-Time compared to Experiment 2, respectively. One interesting pattern in this experiment is that the average number of missed faults decreased significantly compared to Experiment 1 (when S = 0.1 and N = 30%). It is also worth noting that when we compared the results for Joda-Time obtained from Experiment 3 with those from Experiment 2, the number of selected tests reduced by 4% while its fault detection rate increased by almost 5%. This observation confirms that assigning the right value for each parameter would improve both the effectiveness and efficiency of ReTEST.

In Experiment 4, when we set S = 0.25 and N = 75%,

TABLE II: ReTEST vs Safe Technique.

	# Selected Tests		Reduction (% of saved tests)			Fault Detection	
	ReTEST	Safe	ReTEST	Safe	Improvement	# Missed	FDR
Experiment 1: S : 0.1 N = 30%							
P_1	183	439	65	16	58.33	4	85.71
P_2	339	538	42	8	36.95	3	90.62
P_3	1734	3014	58	27	42.46	6	85.36
P_4	1610	3345	61	19	51.85	5	82.14
P_5	701	867	24	6	19.14	1	90.00
P_6	1336	2290	58	28	41.66	1	90.00
P_7	1243	2150	63	36	42.18	2	90.47
P_8	1787	2851	58	33	58.20	3	85.71
P_9	1623	2661	64	41	72.88	4	66.66
P_{10}	1454	3330	69	29	56.33	2	88.88
P_{11}	393	1149	74	24	65.78	2	83.33
P_{12}	193	1032	77	52	52.8	7	30.00
P_{13}	873	1958	63	17	55.42	2	84.61
P_{14}	884	2063	67	23	57.14	1	87.50
P_{15}	1451	2436	47	11	40.44	1	90.90
Experiment 2: S : 0.15 N = 40%							
P_1	230	439	56	16	47.61	2	92.85
P_2	304	538	37	8	31.52	0	100
P_3	1197	3014	71	27	60.27	3	92.68
P_4	1528	3345	63	19	54.32	3	89.28
P_5	673	867	27	6	22.34	0	100
P_6	1495	2290	53	28	34.72	1	90.00
P_7	1444	2150	57	36	32.81	1	95.23
P_8	1744	2851	59	33	38.80	2	90.47
P_9	1308	2661	71	41	50.84	1	91.66
P_{10}	2209	3337	53	29	33.80	1	94.44
P_{11}	589	1149	61	24	48.68	1	91.66
P_{12}	344	1032	84	52	66.66	2	80.00
P_{13}	1014	1958	57	17	48.19	0	100
P_{14}	1045	2063	61	23	49.35	0	100
P_{15}	1615	2436	41	11	33.70	0	100
Experiment 3: S : 0.20 N = 50%							
P_1	256	439	51	16	41.66	1	96.42
P_2	380	538	35	8	29.34	0	100
P_3	1816	3014	64	27	39.72	3	92.68
P_4	2106	3345	56	19	37.03	2	92.85
P_5	692	867	25	6	20.21	0	100
P_6	1431	2290	55	28	37.5	0	100
P_7	1545	2150	54	36	28.12	0	100
P_8	1659	2851	61	33	41.79	1	95.23
P_9	1217	2661	73	41	54.23	0	100
P_{10}	2021	3337	57	29	39.43	1	94.44
P_{11}	695	1149	54	24	39.47	0	100
P_{12}	408	1032	81	52	60.41	2	80.00
P_{13}	1156	1958	51	17	40.96	0	100
P_{14}	1125	2063	58	23	45.45	0	100
P_{15}	1724	2436	37	11	29.21	0	100
Experiment 4: S : 0.25 N = 75%							
P_1	282	439	46	16	35.71	0	100
P_2	409	538	30	8	23.91	0	100
P_3	949	3014	77	27	68.49	2	95.12
P_4	1362	3345	67	19	59.25	2	92.85
P_5	655	867	23	6	24.46	0	100
P_6	1622	2290	49	28	29.16	0	100
P_7	1881	2150	44	36	12.5	0	100
P_8	2042	2851	52	33	28.35	0	100
P_9	1488	2661	67	41	44.06	1	91.66
P_{10}	2115	3337	55	29	36.61	0	100
P_{11}	771	1149	49	24	32.89	0	100
P_{12}	494	1032	77	52	52.08	1	90.00
P_{13}	1227	1958	48	17	37.34	0	100
P_{14}	1205	2063	55	23	41.55	0	100
P_{15}	1779	2436	35	11	26.96	0	100
Experiment 5: S : 0.35 N = 100%							
P_1	224	439	59	16	51.19	3	89.28
P_2	345	538	57	8	53.26	5	84.37
P_3	1073	3014	80	27	72.60	7	82.92
P_4	1404	3345	72	19	65.43	5	82.14
P_5	849	867	29	6	24.46	0	100
P_6	1431	2290	55	28	37.5	1	90
P_7	1377	2150	59	36	35.93	3	85.71
P_8	1191	2851	72	33	58.20	4	80.95
P_9	857	2661	81	41	67.79	3	75.00
P_{10}	1504	3337	73	29	61.97	2	88.88
P_{11}	498	1149	67	24	56.57	1	91.66
P_{12}	190	1032	91	52	81.25	2	80.00
P_{13}	1180	1958	58	17	49.39	0	100
P_{14}	1045	2063	64	23	53.24	0	100
P_{15}	1670	2436	43	11	35.95	1	90.90

the average number of selected tests for nopCommerce and JFreeChart increased by 8% and 0.11% while it decreased by 4% and 30% for Joda-Time and Umbraco compared to Experiment 3. The results of this experiment show that both versions of nopCommerce, 5 out of 6 versions of Joda-Time, and 4 out of 5 versions of JFreeChart reached their maximum fault detection rate. Also the average number of missed faults decreased significantly compared to experiment 1 and 2, which showed that in several versions of each application ReTEST was able to detect 100% of the faults. Moreover, the most interesting outcome of this experiment so far is that the number of selected test cases for Umbraco reduced up to 30% compared to the experiment 3, while its average fault detection rate increased to 5%.

Finally, in Experiment 5, we set S = 0.35 and N = 100%, which indicates that in this experiment, ReTEST prioritized textual similarity of program changes and test case over diversity. As can be seen, by maximizing the similarity value number of missed faults increased and FDR dropped off. The maximum number of missed faults is relatively high. Overall ReTEST was able to detect 100% of the faults, only in three out of fifteen projects. Although, the total number of selected test cases decreased compared to experiment 2 to 4. We speculate the reason of this phenomena is that when we set S = 0.35 there were smaller number of the test cases that did achieve the similarity score 0.35 and above.

Overall, using ReTEST we were able to reduce the number of selected tests up to 60% on average compared to the safe technique. While this reduction was significant, when we set S = 0.25 and N = 0.75, the FDR using ReTEST is only 3.36% less than the safe technique. The percentage of selected test cases reduced up to 36.8% on average for all object programs. These results indicate that our approach can be beneficial and offer great savings when software companies have the limited time budget during regression testing.

B. RQ2 Analysis

To answer RQ2, we selected three different parameter settings for ReTEST, which are Experiments 1, 4, and 5 in Table II, and compared these three experimental results with those of two control techniques (coverage- and diversity-based techniques). We selected these parameter values because for Experiment 1, the diversity is a more important factor than the similarity while for Experiment 5, the importance of these factors is reversed (further, these two experiments are the worst cases for ReTEST). Also, we selected Experiment 4 because its results are more stable compared to others. In this analysis, we collected the results by foreshortening test execution time by 20% for each project.

Figure 6 illustrates the mean percentage of faults detected over time. In this figure, ReTEST-1, 4, and 5 refers to Experiments 1, 4, and 5 of ReTEST. From Figure 6, we can see that for the majority of applications under test (eleven out of fifteen), ReTEST detected higher mean numbers of faults than the other techniques as test execution time increases. However, there are cases where the performance of ReTEST

is not better than other approaches. For example, ReTEST did not perform well compared to Div and Cov for JFreeChart V1.0.3 and Joda-Time V1.4. The numbers of faults detected by Cov and Div are much larger than the number of faults revealed by ReTEST. For JFreeChart V1.0.3, the numbers of faults detected by Cov , Div, and ReTEST-1 are 5, 6, and 2, respectively. In the case of Joda-Time V1.4, ReTEST-1, and ReTEST-4 found 6 faults by executing 20% of tests while Cov and Div detected 10 and 12 faults, respectively. We speculate that the small number of queries resulted in this outcome.

However, the result for the smaller program shows the contrary; the code coverage-based technique (Cov) and ReTEST-5 outperformed the diversity-based technique, ReTEST-1 and ReTEST-4 (e.g, Joda-Time V0.95, and nopCommerce V2.10). The diversity-based technique (Div), which performs well for the larger programs, manages to produce the results that are close to those produced by Cov, however, ReTEST still outperformed compared to them. ReTEST outperformed both control techniques for larger scale projects when a sufficient amount of code change information is available.

V. Discussion

Our results indicate that the effectiveness and efficiency of test selection can be improved through the use of term similarity, test diversity, and historical test results together. As shown in Table II, the use of appropriate parameter settings produced better fault detection and test reduction rates. In this section, we discuss additional observations and implications of our results, and address the limitations of our work.

A. Implications of ReTEST

As we mentioned earlier, modern software technologies tend to grow much faster than before, but often projects' budget and time are limited. To deliver a robust software product, testers usually test the entire system, but this practice is often impractical and infeasible due to the rapid software release cycle (testing the entire system might take several hours while the release speed in an agile system could be less than an hour [27]). ReTEST addresses this limitation by providing a powerful and yet novel data modeling that can reduce the cost of regression testing by selecting test cases based on the test quality measures (e.g., fault history and diversity) and budget requirements. ReTEST is built based on a graph database, which is efficient and very flexible, and thus supports today's agile software delivery practices.

One of the advantages of ReTEST is that its performance remains nearly constant even with the growth of the dataset. This aspect is particularly important for regression testing because the number of tests can grow rapidly as developers release each program version (e.g., the number of test cases of Joda-Time version 1.5 was increased by 409% compared to version 0.95). The performance of ReTEST is stable even when the size of the system grows because each query is localized to a portion of the graph database. As a result, the execution time for each query is proportional only to the size of the part of the graph traversed to extract that query, instead of the size of the entire graph. For example, while ReTEST took 61 seconds to select tests from the graph with 923 nodes (test cases) of Joda-Time V0.95 using 143 queries, it only took 97 seconds to select a set of test cases from 4,701 test cases (5 times larger) of Joda-Time v1.5 for 112 queries.

Handling newly added test cases is another limitation of traditional regression testing techniques. Assume that a set of new test cases has been added but there is no history information about them to measure their quality in terms of previous bug detection history or code coverage. For the traditional selection techniques, handling this situation requires to update and maintain the database through various analyses, which can be very time consuming. ReTEST can address this problem by adding new tests into the graph database and estimating their potential bug detection capability. This can be done by measuring their similarity to the old test cases without disturbing the structure of the existing graph. The schema-free nature of ReTEST helps to simultaneously relate data elements in different ways. This allows ReTEST to update the graph database easily as software evolves, and thus, reduces the maintenance overhead. The results of our experiment show that 31 of 242 newly added tests for Joda-Time v0.99, 214 out of 896 for Joda-Time v1.3, and 43 out of 196 for JfreeChart v1.0.7 revealed faults. These test cases were selected by ReTEST during the test selection process.

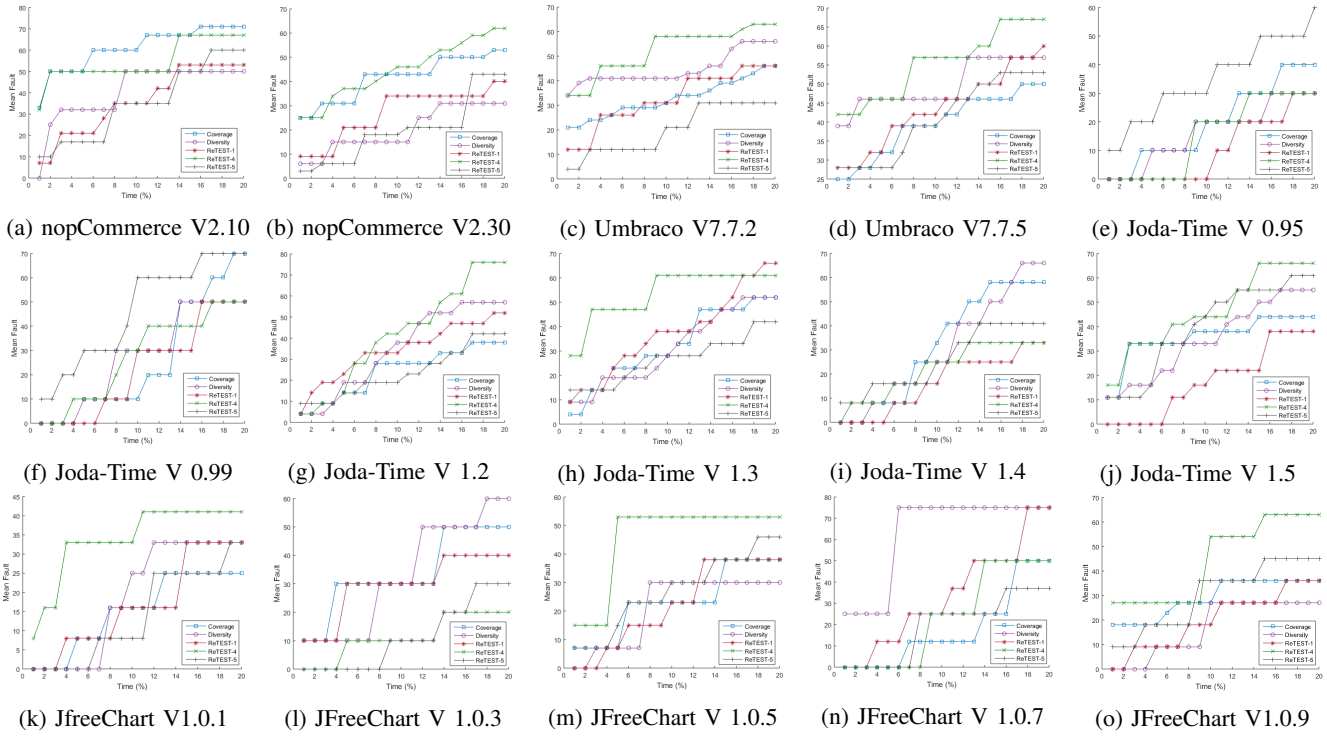
B. Limitations of Applying ReTEST:

While our results indicate that ReTEST can provide an efficient and flexible solution for test selection, it also has some limitations. In our study, we applied a term similarity measurement to estimate how much each test can cover the changes in a file. This means that our results are highly dependent on the accuracy of term similarity calculation. For example, when we compared versions 7.7.0 and 7.7.2 of Umbraco, we found that the fault revealing test method was:

```
public void RewritePath_Exceptions
(string input,string path,Type exception){
var source = new Uri(input);
Assert.Throws(exception,()=>{
var output = source.Rewrite(path)});}
```

As this test method shows, most of the words in this test case are general terms that can be found in any test, thus when we extracted a query, this test had been neglected by ReTEST for selection. However, our overall results show that the probability of such occasions is very low.

Another limitation involves the choice of parameter values. As we observed from the results, parameter settings can affect the outcome of ReTEST. We speculated that this outcome was affected by each program's characteristics (e.g., the number of lines of code changed, the number of test cases, similarity values among test cases, etc). For instance, nopCommerce v2.30 has 585 test cases, and the number of queries that we built for the version pair 2.20 and 2.30 is 118. This means that the ratio of the number of tests to the number of queries is 4.94, which is very low (the average rate of tests per query in our study is 26.75). For this case, when we set S = 0.15, we obtained FDR = 100 by running only 304 test cases, while when we sat S = 0.35, we obtained FDR = 84 by running



(a) nopCommerce V2.10 (b) nopCommerce V2.30 (c) Umbraco V7.7.2 (d) Umbraco V7.7.5 (e) Joda-Time V 0.95

(f) Joda-Time V 0.99 (g) Joda-Time V 1.2 (h) Joda-Time V 1.3 (i) Joda-Time V 1.4 (j) Joda-Time V 1.5

(k) JfreeChart V1.0.1 (l) JFreeChart V 1.0.3 (m) JFreeChart V 1.0.5 (n) JFreeChart V 1.0.7 (o) JFreeChart V1.0.9

Fig. 6: Mean percentage of faults detected over time, where time is the percentage of total execution time of all the test cases.

345 test cases. This indicates that the diversity works better when the ratio of the number of tests to the number of queries is smaller. In another case, when we extracted queries for the version pair 1.3 and 1.4 in Joda-Time, we built only 54 queries, and by setting S = 0.2, we were able to find 100% of faults by running 1217 test cases. But, when we sat S = 0.1, the percentage of faults detected decreased to 66% by running 1623 test cases. This means that a slight increase in the similarity score can improve the efficiency as well as the effectiveness of ReTEST.

From our results, we also observe that ReTEST performs best when applications have a large amount of code changes. For instance, graphs (l) and (i) in Figures 6 show that the performance of ReTEST drops significantly when the numbers of queries are small (e.g., the numbers of queries for JFreeChart 1.0.2-1.0.3 and Joda-Time 1.3-1.4 are 6 and 54, respectively). From these observations, we can conclude that we need to select appropriate parameter values for ReTEST considering the number of queries, the number of tests, and the diversity among tests. We plan to improve the ReTEST performance by adding a decision maker module that can suggest the best parameter values based on each program's characteristics.

C. Cost-Benefit Analysis:

As we discussed earlier, we were able to reduce the cost of test selection by applying a lightweight technique and eliminating the coverage profiling cost. However, the savings demonstrated do not guarantee the cost effectiveness because the techniques also have associated costs (e.g., initial setup cost, analysis cost, etc.) To evaluate which technique is better in given situations, we discuss the cost-benefits of ReTEST and other control techniques. To perform the cost-benefit analysis, we divided the costs for applying ReTEST in two categories: 1) initial cost, and 2) incremental cost. The initial cost in this study is the cost of data construction (e.g., building the graph database and setting properties values), and the cost of tool implementation. Once we collect the required data and build the tool, we do not need to repeat this process again.

The incremental cost in this study includes updating the graph database (adding new nodes, edges, and their properties) and building new queries. This cost varies on code changes. The incremental cost for most applications is negligible. For instance, the worst case scenario is updating the database from version 0.95 to 0.99 of Joda-Time (because the number of tests between these two versions increased by 244%, we also needed to extract 128 queries from the database). For this specific case, it took only 14 seconds to update the database (from 923 to 3,181 tests) and 58 seconds for query extraction.

The main costs of two control techniques (code coverage and diversity) include: 1) instrumentation, 2) test execution, and 3) algorithm execution. All above mentioned costs belong to the incremental cost category (because coverage information has to be updated for each new release). In the best case scenario for nopCommerce v2.10, the total cost of test selection using the code coverage technique is 33.07 minutes (the number of test cases in nopCommerce v2.10 is smaller than other projects). This cost includes 28.35 minutes for profiling, 4.66 minutes for test execution and 4 seconds for calculation (sorting tests based on their total code coverage). The cost of the diversity-based technique for the same projects requires the same amount of time for profiling and test execution as

the code coverage technique does include additional 8 seconds for calculating diversity among the test cases. However, using ReTEST for the same program, the total selection cost is 14 seconds, which includes the costs for adding 5 new nodes to the graph database (3 seconds) and extracting 65 queries (11 seconds). Thus, using ReTEST, we reduced the cost up to 14092% compared to two other techniques.

The safe technique is the most expensive technique among others: the complexity of this algorithm is $O(Tnn')$, where n and n' are a pair of nodes of programs G and G', and T is a given test suite [42]. This technique also requires instrumentation and test execution, which can be very expensive for large scale applications with a large number of tests.

Considering the cost-benefit tradeoffs among techniques that we discussed so far, we believe that ReTEST would offer great savings in general. Further, if we apply ReTEST to large scale industrial applications in which test execution might take several days (e.g., mySql, Chrome, etc) [28], ReTEST will provide even greater savings.

VI. Threats to Validity

The main threat to validity is the choice of the similarity value and the number of test cases to be selected, which can affect the results. In this experiment, we used multiple similarity values ($0.1 < S < 0.35$) and different percentages of test cases to be selected ($30\% < N < 100\%$) to examine the tradeoffs between these parameter values. Because our proposed algorithm is flexible, these values can be easily adjusted depending on available time and resources, program characteristics (e.g.,, the number of lines of code changed, the number of test cases, etc), or organization preferences.

VII. Related Work

Reducing the cost of regression testing has been an active research topic because it is expensive and time consuming. Numerous regression testing techniques have been proposed to provide cost-effective solutions considering various types of data sources (e.g., code coverage, test case diversity, and fault history) [17], [23], [47], and the code coverage-based technique is one of the most widely used and evaluated techniques. Although the code coverage-based approach is naïve and easy to implement, many empirical studies have shown that it can be effective [19], [20], [22], [40]. Recently, other types of data sources and software artifacts (e.g., code complexity, test diversity, fault history, and requirements documents) have also been utilized to implement regression testing techniques, but here, we limit our discussion to test diversity- and fault history-based techniques that are most closely related to our work.

Researchers have used the test diversity to improve testing techniques because more diverse tests tend to cover a wider range of program code areas [25], [26], [41]. For example, Hemmati et al. [25] investigated possible similarity functions to support similarity-based test selection in the context of model-based testing, using genetic algorithms to examine an industrial software system. Rogstad et al. [41] investigated the use of diversity-based test case selection for black-box testing of database applications. Mondal et al. [35] proposed a test selection approach that maximizes the code coverage and diversity among selected test cases using NSGA-II multi-objective optimization. The results of their study showed that the use of two types of information (code coverage and diversity) is more effective than single objective approaches.

In fault history-based techniques, test cases that have failed in previous releases have a high priority during the selection process. Previous studies have shown that test cases with fault detection history are more likely to fail in future versions [30], [38], [46]. For example, Anderson et al. [12] showed that the most frequent failures from the history information is a good indicator for future failure predictions. Kim and Porter [30] also showed that prioritizing test cases based on the test execution history can improve the effectiveness of regression testing. More recently, Noor and Hemmati [38] defined a set of metrics that estimate the quality of tests using their similarity to the previously failed tests.

Although aforementioned techniques could improve the effectiveness of regression testing in different ways, they all depend on either dynamic or static code coverage information. Considering the increasing demands on rapid development and release cycles, the techniques that require expensive application costs would not be suitable for modern software development practices. Some researchers have investigated ways to reduce the costs and overhead of regression testing techniques. For example, Saha et al. [43] proposed a technique that maps the traditional regression testing problem to an information retrieval problem They used the differences between two program versions as a query and a test suite as a document, and then prioritized test cases based on their cosine similarities to the queries.

Unlike aforementioned techniques, ReTEST considers multiple metrics for test selection, which makes it flexible and more effective. Moreover, ReTEST addresses the limitations of fault history- and coverage-based techniques by extracting program information from a graph database rather than performing coverage profiling.

VIII. Conclusions and Future Work

In this research, we have introduced a novel technique, ReTEST, which is lightweight and language-independent, to improve the effectiveness of regression testing. Our empirical results indicate that ReTEST can reduce the cost of regression testing considerably. Because ReTEST is built based on a graph database and also it eliminates coverage profiling, it can be aligned with the today's agile software delivery practices in which traditional regression testing techniques might not be suitable. For future work, we wish to continue expanding this research by adding a decision maker component that can help testers to assign appropriate parameter values to increase the effectiveness of test selection in terms of both fault detection rate and test reduction. We also plan to investigate how our approach applies to different areas of regression testing such as test prioritization and reduction.

Acknowledgment

This work was supported, in part, by NSF CAREER Award CCF-1564238 to University of North Texas.

REFERENCES

[1] hhttp://meldmerge.org//. [Accessed: Jan. 26, 2017].

[2] hhttp://www.nopcommerce.com/. [Accessed: Jan. 26, 2017].

[3] https://umbraco.com/. [Accessed: Oct. 06, 2017].

[4] https://github.com/rjust/defects4j. [Accessed: Jan. 25, 2018].

[5] https://github.com/nopSolutions/nopCommerce/issues//. [Accessed: Oct. 06, 2017].

[6] http://issues.umbraco.org/issues//. [Accessed: Oct. 06, 2017].

[7] http://www.eclemma.org/jacoco/trunk/doc/maven.html/. [Accessed: Feb. 08, 2018].

[8] K. K. Aggrawal, Yogesh Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. In *ACM SIGSOFT Software Engineering Notes*. ACM, 2004.

[9] A. Aizawa. An information-theoretic perspective of tfidf measures. *Information Processing and Management*, 39(1):46–65, 2033.

[10] M. Al-Hajjaji, Th. Thm, J. Meinicke, M. Lochau, and G. Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the International Software Product Line Conference* , 2014.

[11] J. Anderson, H. Do, and S. Salem. Customized regression testing using telemetry usage patterns. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016.

[12] J. Anderson, S. Salem, and H. Do. Improving the effectiveness of test suite through mining historical data. In *Working Conference on Mining Software Repositories (MSR)*, 2014.

[13] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. *Manifesto for agile software development*. 2007.

[14] D. Binkley and D. Lawrie. Applications of information retrieval to software maintenance and evolution. In *Proceedings of ESE*, 2010.

[15] R. Carlson, H. Do, , and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *ICSM '11 Proceedings of the 2011 27th IEEE International Conference on Software Maintenanc*, pages 382–391. IEEE-ACM, 2011.

[16] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto. On the use of a similarity function for test case selection in the context of model-based testing. In *Proceedings of the International Conference of Software Testing, Verification and Reliability*. IEEE-ACM, 2009.

[17] C. Catal and D. Mishra. Test case prioritization: a systematic mapping study. 21(3):445–478, 2013.

[18] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Conference on Mining Software Repositories (MSR)*, pages 31–41. IEEE, 2010.

[19] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. IEEE-ACM, 2008.

[20] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. 36(5), 2010.

[21] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*, pages 329–338, May 2001.

[22] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[23] E. Engstrom, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[24] M G. Epitropakis, Sh. Yoo, M. Harman, and E.K. Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, pages 234–245, 2015.

[25] H. Hemmati, A. Arcuri, and L. C. Briand. Empirical investigation of the effects of test suite properties on similarity based test case selection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2011.

[26] H. Hemmati, A. Arcuri, and L. C. Briand. Achieving scalable model based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, (1):1–43, 2013.

[27] H. Hemmati, Z. Fang, and M. V. Mantyla.

[28] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, April 2014.

[29] J. Jenkins. Velocity culture (the unmet challenge in ops). In *Presentationat O'Reilly Velocity Conference*. IEEE, 2011.

[30] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[31] J.H Kwon, I.Y Ko, G. Rothermel, and Staats M. Test case prioritization based on information retrieval concepts. In *Asia-Pacific Software Engineering Conference*, 2014.

[32] M. Lease, J. Allan, and W. B. Croft. Regression rank: Learning to meet the opportunity of descriptive queries. In *ECIR*, pages 90–101. IEEE, 2009.

[33] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Journal of Automated Software Engineering.*, 19(1):65–95, 2012.

[34] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[35] D. Mondal, H. Hemmati, and S. Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *International Conference in Software Testing Verification and Validation (ICST)*. IEEE, 2015.

[36] L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, and A. Marcus. Query-based configuration of text retrieval solutions for software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[37] C. D. Nguyen, A. Marchetto, and P. Tonella. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *Proceedings of ICWS*, 2011.

[38] T. Bin Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *International Conference in Software Reliability Engineering (ISSRE)*. IEEE, 2015.

[39] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, May 1988.

[40] X. Qu, M.B. Cohen, and G.Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 75–85. IEEE-ACM, 2008.

[41] E. Rogstad, L. C. Briand, and R. Torkar. Test case selection for blackbox regression testing of database applications. *Information and Software Technology*, 55(10):1781–1795, 2013.

[42] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[43] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *International Conference on Software Engineering (ICSE)*, pages 268–279. IEEE-ACM, 2015.

[44] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. In *Empirical Software Engineering*, pages 1–31. IEEE-ACM, 2012.

[45] R. Wang, Sh. Jiang, and D. Chen. Similarity-based regression test case prioritization. In *Proceedings of the Software Reliability Engineering (ISSRE)*, 2015.

[46] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 140–150, July 2007.

[47] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation : A survey. *Software Testing, Verification, and Reliability*, March 2010.

[48] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[49] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of ICSE*, 2013.

[50] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *International Workshop on Predictor Models in Software Engineering, PROMISE*. IEEE, 2007.