

# QRTTest: Automatic Query Reformulation for Information Retrieval Based Regression Test Case Prioritization

Maral Azizi

Departemant of Computer Science  
East Carolina University  
azizim19@ecu.edu

**Abstract**—The most effective regression testing algorithms have long running times and often require dynamic or static code analysis, making them unsuitable for the modern software development environment where the rate of software delivery could be less than a minute. More recently, some researchers have developed information retrieval-based (IR-based) techniques for prioritizing tests such that the higher similar tests to the code changes have a higher likelihood of finding bugs. A vast majority of these techniques are based on standard term similarity calculation, which can be imprecise. One reason for the low accuracy of these techniques is that the original query often is short, therefore, it does not return the relevant test cases. In such cases, the query needs reformulation. The current state of research lacks methods to increase the quality of the query in the regression testing domain. Our research aims at addressing this problem and we conjecture that enhancing the quality of the queries can improve the performance of IR-based regression test case prioritization (RTP). Our empirical evaluation with six open source programs shows that our approach improves the accuracy of IR-based RTP and increases regression fault detection rate, compared to the common prioritization techniques.

**Keywords:** Regression Testing, Test Case Prioritization, Software Repository, IR-based Regression Testing, Query Reformulation.

## I. INTRODUCTION

In continuous integration development (CI), the source code of a system frequently changes due to bug fixes or new feature requests. Some of these changes may accidentally cause regression issues to the newly released software version. Because of the significant code churn aspect of CI systems, automated testing plays an essential part of these systems to ensure the correctness of the newly released software [24]. Due to the limited testing time and budget, often practitioners select a number of the test cases that may potentially exercise the portion of the code that has been changed. However, finding the dependency between test cases and code churn often requires static or dynamic code analysis. These testing techniques often require collecting multiple sources of information from source code such as code coverage, fault history, code complexity, etc. Although, many empirical studies have shown the effectiveness of these techniques, they suffer from several limitations, 1) this information is usually unavailable before test cases are executed, 2) collecting this information requires a significant amount of time and human knowledge,

3) collecting information is a continuous task, which has to be repeated for each new version of a software, 4) previously collected information is not accurate for a new version of a program due to the changes in source code and test suite, and 5) performing code analysis is often costly and time consuming [10], [30].

More recently, some researchers proposed information retrieval to develop techniques for automated regression testing [36], [10], [39]. These techniques are instances of text retrieval concept in source code and traceability link recovery. IR-based techniques rely on formulating a query based on the code artifacts to extract meaningful information from the source code. Although IR-based techniques have received significant attention in solving a variety of software engineering problems, queries are often short and contain few key phrases, which can lead to low recall and often prone to errors and omissions [13], [14]. In the standard IR-based regression test case prioritization (RTP) technique, a query is simply a vector of words that have been extracted from the code commits, which usually contains non-discriminatory terms that can be found in a vast number of source files; therefore, a retrieval algorithm often does not return the relevant test cases. In such cases, the query is considered low-quality. Some researchers have suggested reformulating the low-quality queries by offering various techniques, e.g. incorporating feedback from users, utilizing feedback from previous search results, expanding the query by adding more information, and reducing the size of the query [21], [14], [26], [37]. However, to date no little study has been done to review the effectiveness of query reformulation in the regression testing domain.

To address the above mentioned limitations, in this paper, we propose an automated regression test case prioritization approach that utilizes automated query reformulation for information extraction that we name QRTTest. QRTTest receives the code churn as a query, builds the initial query, then reformulates the query to increase the effectiveness of information extraction. Finally, QRTTest search algorithm ranks the test cases that exercise the most recent changes in the source code base on the proposed ranking algorithm. This paper is the first study of automated query reformulation in RTP that describes all processing steps, discusses the retrieval performance, identifies the limitations, and suggests alternative

solutions for the limitations of IR-based RTP.

To evaluate QRTTest, we used 37 versions of 6 open-source software projects written in Java and C# and compared our results against five widely used test case prioritization techniques. The experimental results show that QRTTest can be effective in practice by reducing a significant amount of time compared to the common regression testing techniques such as code coverage-based techniques. The main contributions of this research are as follows:

- We introduce a new cost effective regression test case prioritization technique. Our proposed technique is fast, scalable, and light-weight, which eliminates the cost of coverage profiling.
- Our proposed approach provides an effective query reformulation technique that increases the retrieval accuracy as well as regression testing performance.
- Our study also empirically evaluates the proposed approach by comparing it with five test prioritization techniques that are commonly used for regression testing.

The rest of the paper is structured as follows. Section II explains the QRTTest approach. Section III outlines the research questions and the details of the experimental design. Section IV presents the results of our study. Section V discusses the results including practical implications. Section VI discusses the threats to validity. Section VII presents related work, and Section VIII discusses conclusions and future work.

## II. APPROACH

### A. Overview of the Approach

Figure 1 presents an overview of the proposed technique, which includes four major steps: document construction, query construction, query reformulation, and retrieval. First, QRTTest uses tests source files to construct documents. To build the document collection, QRTTest builds the abstract syntax tree (AST), preprocess the files, index them and store the indexed elements into a database (the details are explained in Section II-B). 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 the program, which can increase the chances of fault detection. We then reformulate the query by adding relevant terms to increase the effectiveness of retrieval. In the following subsections, we describe our proposed approach in detail including the query reformulation and retrieval process.

### B. Document Construction

This step transforms the raw data into a format that will be more effective for the retrieval process. Document construction often consists of feature extraction, followed by the construction of an appropriate data structure that is suitable for retrieval and indexing. Constructing documents usually depends on the information granularity and the choice of IR technique. Unit tests are often collections of source files, where each file consists of one or more test classes and test methods. Prior researches on IR-based RTP showed that the lower granularity returns higher accuracy in terms of fault detection [36], [18],

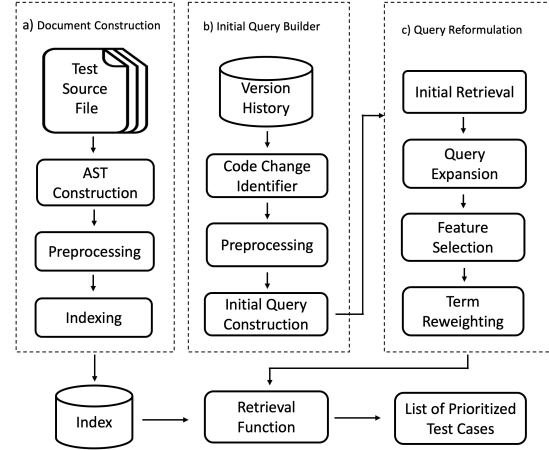


Figure 1: An Overview of QRTTest Framework

[19]. Therefore, we construct the documents in test method level of granularity.

To build the documents, in this work, we first collect all test suite files from each program version. We then build the abstract syntax tree of each file. After building the AST, we extract the methods using Eclipse Java Development Tools (JDT). Following that we tokenize each term in each method and remove stop words, mathematical operators, and all symbols. We also do not remove comments or identifier names because studies showed that they are particularly important from the information retrieval point of view [36], [10].

### C. Initial query construction

The goal of IR systems is to provide a list of the most relevant documents for a given query. When a query contains topic-specific keywords, the accuracy and recall of the system is high; however, queries are often short and contain few key phrases, which can lead to low recall and often prone to errors and omissions. One of the most commonly used and successful techniques to improve the document retrieval is to expand the original query with terms that are related to the documents containing answers to it [14]. To increase the recall and precision in this context we apply two techniques to reformulate the queries. First, we expand the query by adding more relevant key phrases then we assign weights to each token based on their relevance to the initial query.

Box b) of Figure 1 illustrates the process of building the initial query. In this step, first, we compare two versions of the program then we apply a *Diff* tool [1] to identify all changes in the source code. Those files that have been changed from the previous version will be stored. Next, we preprocess those files by removing the stop words and tokenization. Note that stop words in this context are different from standard English stop words. In this research, we remove the most frequent reserved terms in Java and C# languages (i.e. public, private, if, else, etc.). We also remove all symbols including mathematical signs, parentheses, brackets, etc. After the preprocessing is

complete, we build the initial query, which is a vector of tokens.

#### D. Query Reformulation

As we discussed earlier, in order to increase the effectiveness of retrieval we need to enrich the quality of the queries, which is known as Query Reformulation (QR). To date, many researchers in the software engineering community have investigated a variety of QR techniques [15], [31], [21]. Some of these techniques require developers to look into the query and manually select the appropriate terms to expand the query, however, this will add an extra burden on the software developer. Another technique is using pseudo-relevance feedback, which is an automated technique for QR. In this technique, after receiving the initial set of retrieval results, some of the features from the top retrieved documents will be selected for the query expansion. The underlying hypothesis of this technique is that since the selected features have the highest retrieval scores, they are likely to contain further informative terms that are related to the original query [13]. In this study, we apply pseudo-relevance feedback for QR, which is automated and eliminates the need for user inputs. The algorithm is explained in the following steps:

1) *Initial Retrieval*: In this step, an initial list of ranked documents for a given query is obtained by utilizing cosine similarity metric.

2) *Query Expansion*: In order to expand the query in this study we use the Relevance Model (RM) technique. This technique incorporates the ranking scores of the documents  $D$  as an additional component for a more accurate selection of the terms to be used for reformulating the original query. This technique aims to calculate the probability of co-occurrence of term  $t$  in document  $D$  with the terms in  $Q$ . RM estimates this probability by below formula:

$$P(t|Q) \simeq \sum_{d \in D} P(t|d) P(d) \prod_{i=1}^m P(q_i|d)$$

where  $Q = q_1, q_2, \dots, q_m$  is a query,  $q_i$  is the  $i$ -th term in  $Q$ ,  $P(t|d)$  is the probability of the term  $t$  for a given document  $d$ , and  $D$  is a document set. The query likelihood component in the above formula,  $\prod_{i=1}^m P(q_i|d)$ , returns the similarity score of the document  $d$  with respect to the original query.

3) *Selection of Expansion Features*: After ranking the candidate documents, the top terms will be selected for query expansion. Often the selection is made based on the retrieval's objectives, regardless of the dependencies between the selected features. To ensure the computational efficiency, usually a limited number of terms is selected for the expansion. The typical choice is to use 10-30 features [13]. In this study, we select the top 10 features to expand the original query.

4) *Term Re-weighting*: A common practice in term re-weighting is to give more importance to the original query

terms. Subsequently, smoothing the probability  $P(t|Q)$  by the collection probability of the candidate term  $P_c(t)$  gives:

$$P_s(t|Q) = \lambda \cdot P(t|Q) + (1 - \lambda) \cdot P_c(t)$$

where  $P_s(t|Q)$  is a smoothed probability and  $\lambda$  is a smoothing parameter.  $\lambda$  is an experimental parameter that determines to what extent one should mix the original query with new terms by their associated probabilities. The value of  $\lambda$  can be adjusted based on the data. In this study, we assigned  $\lambda = 0.8$  to examine the effectiveness of expansion terms. Section V-A describes the effects of different  $\lambda$  values on the retrieval accuracy.

#### E. Final Retrieval

Assume that documents and a query are shown as below:

$$D = \{d_1, d_2, d_3, \dots, d_n\}$$

$$Q = \{q_1, q_2, q_3, \dots, q_n\}$$

where each element  $i$  in  $D$  and  $Q$  represent the term frequency of  $term_i$  in document  $D$  and query  $Q$ , respectively. Further, to obtain the final results the expanded query is used to retrieve the data from the set of documents using the below formula:


$$Sim(D, Q) = \sum_{i=1}^n tf_D(d_i) tf_Q(q_i) idf(t_i)^2$$

where  $tf_D(d_i)$  is the term frequency of  $term_i$  in the document  $d$ ,  $tf_Q(q_i)$  is the term frequency of  $term_i$  in the query  $q$ , and  $idf(t_i)$  is the inverse document frequency of  $term_i$  in the collection.

#### F. An Example of Query Construction and Reformulation

Figure 2 shows an example of the query construction process. Top section of the figure represent a code commit of *Joda-Time* program<sup>1</sup>. The bottom section shows the initial query vector that is constructed by following the instruction explained in Section II-C. Each query is represented as a vector of tokens with a probability value. The original query contains five initial key words ("yeartouse", "month", "imax", "math", "signum") that we stemmed each word to its root form (e.g. months to month). The two common words *if*, *else*, and the mathematical symbols are also filtered. The other 10 tokens that are shown in the bottom layer are added to enrich the quality of the query since it is very short. The initial tokens of the query vector are extracted from the body of the code commit using *Tf-Idf* formula and the recommended features for the expansion are extracted from the relevant documents using *RM* technique.

<sup>1</sup>source code and test cases of this program can be found at <https://github.com/JodaOrg/joda-time>

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

112	112	<code>// Initially, monthToUse is zero-based</code>
113	113	<code>int monthToUse = thisMonth - 1 + months;</code>
114	114	<code>if (thisMonth &gt; 0 &amp;&amp; monthToUse &lt; 0) {</code>
115	-	<code>yearToUse++;</code>
116	-	<code>months -= iMax;</code>
115	+	<code>if (Math.signum(months + iMax) == Math.signum(month))</code>
116	+	<code>{</code>
117	+	<code>yearToUse--;</code>
117	+	<code>months += iMax;</code>
118	+	<code>} else {</code>
119	+	<code>yearToUse++;</code>
120	+	<code>months -= iMax;</code>
121	+	<code>}</code>
117	122	<code>monthToUse = thisMonth - 1 + months;</code>
118	123	<code>}</code>
119	124	<code>if (monthToUse &gt;= 0) {</code>
119	124	<code>}</code>
Initial Query Vector		<code>{&lt;"yeartouse", 0.19&gt;, &lt;"month", 0.27&gt;, &lt;"imax", 0.17&gt;, &lt;"math", 0.22&gt;, &lt;"signum", 0.13&gt;}</code>
Recommended Features		<code>{&lt;"time", 0.21&gt;, &lt;"year", 0.18&gt;, &lt;"monthouse", 0.066&gt;, &lt;"day", 0.086&gt;, &lt;"timezone", 0.12&gt;, &lt;"datetime", 0.13&gt;, &lt;"chronology", 0.092&gt;, &lt;"min", 0.079&gt;, &lt;"daytouse", 0.11&gt;, &lt;"datetimefiledtype", 0.14&gt;}</code>

Figure 2: An Example of Query Construction

### III. EMPIRICAL STUDY

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

- RQ1: Is QRTest effective in identifying relevant test cases to the changed files?
- RQ2: How does QRTest perform compared to the common prioritization techniques in terms of fault detection?

#### A. Objects of Analysis

In this study, we used six 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 method level), “TstClass” (the number of test cases in class level), “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, which has been written in C# language [3]. Four other applications were obtained from the defects4j database [4]: *JFreeChart*, *Joda-Time* and *Commons-Lang*, which are written in Java. All the faults for *JFreeChart*, *Joda-Time* and *Commons-Lang* are real, reproducible, and have been isolated in different versions. The faults used in *nopCommerce* and *Umbraco* have been reported by users <sup>2</sup>.

#### B. Variables and Measures

1) *Independent Variables*: The independent variable in this study is regression test prioritization technique. We considered five test case prioritization techniques, which we classified them into two groups: control and heuristic. For our heuristic

<sup>2</sup>The reported faults for *nopCommerce* are available in the *GitHub* repository [5], and the bug history for *Umbraco* is accessible through their website [6].

techniques, we use the approach explained in Section II, therefore, here, we only explain the control techniques.

- 1) Untreated ( $T_1$ ): This technique executes test cases based on their original order.
- 2) Random ( $T_2$ ): This technique prioritizes test cases randomly. We repeat this technique 30 times for each program.
- 3) Total Statement ( $T_3$ ): This technique prioritizes test cases based on the total number of statements they cover. If multiple tests cover the same number of statements, they are ordered randomly.
- 4) Additional Statement ( $T_4$ ): This technique prioritizes test cases based on the number of additional statements they cover. If multiple test cases cover the same number of statements not yet covered, they are ordered randomly.
- 5) Text Retrieval Based ( $T_5$ ): This technique applies standard IR to prioritizes test cases. To measure the correlation between test cases and code change, in this technique we use the tf-idf formula. For further details about the implementation of this technique, please refer to Saha et al. [36].

2) *Dependent Variable and Measures*: Dependent variables for RQ1 are precision, recall and F-Measure. For RQ2 the dependent variable is average percentage of fault detection (APFD), which measures the average percentage of faults detected during the test suite execution. The range of APFD is from 0 to 100; higher values indicate faster fault detection rates. Given  $T$  as a test suite with  $n$  test cases and  $m$  number of faults,  $F$  as a collection of detected faults by  $T$ , and  $TF_i$  as the first test case that catches the fault  $i$ , we calculate APFD [35] as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + \dots + TF_m}{nm} + \frac{1}{2n}$$

#### C. Data Collection and Experimental Setup

As described earlier, QRTest does not require any dynamic or static code coverage information and it calculates the similarity among code commits and source code/test cases. To collect code coverage information for control techniques we used Visual Studio Test Analysis plugin for *nopCommerce* and *Umbraco-CMS*. For other four Java applications, we used JaCoCo plugin on netbeans IDE [7]. We also collected test execution time using a PC with CPU Core i7, memory 16 GB, and operating system Windows 10.

### IV. DATA AND ANALYSIS

#### A. RQ1 Analysis

In RQ1, we evaluate QRTest ability to see whether it can accurately detect the relationship between the queries and the test cases. To evaluate the effectiveness of QRTest, we used common accuracy indicators to determine the accuracy of our model. The three accuracy indicators that we used are Precision, Recall, and F-Measure. In order to calculate the accuracy results, we first manually label the test cases to the relevant files that have been changed. Table II shows

Table I: Subject Applications Properties

No	Object	Version pair	LOC	# TstMethod	# TstClass	# Faults	# Queries	# SrcToken	# TstToken
$P_1$	nopCommerce	2.00 - 2.10	133,862	523	126	28	65	17,803	2,768
$P_2$	nopCommerce	2.20 - 2.30	141,232	585	136	32	118	18,052	2,893
$P_3$	Umbraco-CMS	7.7.0 - 7.7.2	312,511	6,025	628	41	97	26,700	14,552
$P_4$	Umbraco-CMS	7.7.4 - 7.7.5	312,886	6,026	628	28	131	26,811	14,546
$P_5$	Joda-Time	0.9 - 0.95	17,818	923	80	10	143	3,328	2,767
$P_6$	Joda-Time	0.98 - 0.99	20,317	3,181	218	10	128	4,905	3,765
$P_7$	Joda-Time	1.1 - 1.2	22,149	3,360	237	21	130	5,290	4,368
$P_8$	Joda-Time	1.2 - 1.3	24,001	4,256	254	21	115	5,380	4,622
$P_9$	Joda-Time	1.3 - 1.4	25,292	4,511	263	12	54	5,496	4,812
$P_{10}$	Joda-Time	1.4 - 1.5	25,795	4,701	266	18	112	5,610	5,106
$P_{11}$	JFreeChart	1.0.0 - 1.0.1	95,669	1,512	225	12	98	14,219	6,039
$P_{12}$	JFreeChart	1.0.2 - 1.0.3	101,713	2,151	231	10	6	11,855	4,750
$P_{13}$	JFreeChart	1.0.4 - 1.0.5	74,434	2,360	351	13	101	12,057	5,102
$P_{14}$	JFreeChart	1.0.6 - 1.0.7	76,065	2,680	366	8	85	12,662	5,522
$P_{15}$	JFreeChart	1.0.8 - 1.0.9	81,541	2,738	393	11	186	12,738	5,636
$P_{16}$	Commons-Lang	3.02 - 3.03	19,705	1,698	83	17	164	6,732	3,492
$P_{17}$	Commons-Lang	3.03 - 3.04	21,132	1,703	83	14	123	7,256	3,866
$P_{18}$	XStream	1.30 - 1.31	15,983	885	150	10	156	3,740	2,563
$P_{19}$	XStream	1.31 - 1.40	16,280	924	140	13	178	3,824	2,662
$P_{20}$	XStream	1.41 - 1.42	16,851	1,200	157	11	206	4,103	2,893

Table II: Retrieval accuracy QRTTest vs Standard IR

Technique	QRTTest			Standard IR		
Object	Precision	Recall	F-Measure	Precision	Recall	F-Measure
$P_1$	0.4366	0.5562	0.4891	0.2934	0.3457	0.3174
$P_2$	0.5997	0.6414	0.6198	0.3517	0.4187	0.3822
$P_3$	0.3118	0.4006	0.3506	0.2426	0.2922	0.2650
$P_4$	0.388	0.5171	0.4433	0.2603	0.3792	0.3086
$P_5$	0.2646	0.3422	0.2984	0.203	0.2803	0.2354
$P_6$	0.1858	0.2509	0.2134	0.1377	0.2023	0.1638
$P_7$	0.1787	0.2437	0.2061	0.1142	0.1847	0.1411
$P_8$	0.1691	0.2376	0.1975	0.1081	0.1812	0.1354
$P_9$	0.1444	0.2216	0.1748	0.0802	0.1632	0.1075
$P_{10}$	0.1591	0.236	0.1900	0.0938	0.1575	0.1175
$P_{11}$	0.1812	0.2741	0.2181	0.1382	0.2019	0.1640
$P_{12}$	0.1452	0.2469	0.1828	0.0743	0.1544	0.1003
$P_{13}$	0.1658	0.2577	0.2017	0.0921	0.1658	0.1184
$P_{14}$	0.1488	0.2251	0.1791	0.0871	0.1576	0.1121
$P_{15}$	0.1406	0.2078	0.1677	0.0816	0.1512	0.1059
$P_{16}$	0.2261	0.3774	0.2827	0.1865	0.2541	0.2151
$P_{17}$	0.2157	0.3502	0.2669	0.1773	0.2522	0.2082
$P_{18}$	0.2462	0.2901	0.2663	0.3117	0.3855	0.3446
$P_{19}$	0.2596	0.3362	0.2929	0.315	0.3862	0.3469
$P_{20}$	0.2062	0.2396	0.2216	0.2884	0.3619	0.3209

the average results for each application. Based on the results in Table II, we conclude that QRTTest, in general, leads to significant improvements over the standard IR technique. The improvements obtained by QRTTest approach are noticeably higher than those obtained by the standard IR method particularly in case of *nopCommerce* and *Umbraco-CMS*, which are larger than other applications ( $\sim 76\%$  Recall increase on average). Looking at the sizes of the query sets, as presented in Table I, we observe that QRTTest is able to improve the performance of 66% of the queries for a total of 1,584 out of 2,396 queries. From the results of our study, we can see that when the performance of the original query is well, the query reformulation is likely to preserve the retrieval accuracy rather than to decrease it on average. Therefore, we can conclude that query reformulation is most useful when the initial query is low quality.

### B. RQ2 Analysis

Our second research question considers whether the application of QRTTest can improve the rate of fault detection for unit

test cases applied to our subject programs. Figure 3 represents the average percentage of the fault detected for each program. The horizontal axis of each figure shows the prioritization techniques and the vertical axis shows the percentage of the mean APFD values for each program. This experiment was performed 30 times with random selections each time. APFD values were computed by prioritizing all test cases for each application.

From the results shown in Figure 3 we can see some variation across applications. In the case of *nopCommerce*, QRTTest outperformed all other techniques. Among all examined control techniques, in overall, the additional coverage technique ( $T_4$ ) is slightly more effective than the others. In the case of *Umbraco-CMS*, the trend is similar to that of *nopCommerce*, but the difference between control and heuristic techniques is more noticeable; QRTTest particularly performed well in case of *Umbraco-CMS*. We speculate that this could be due to the larger number of test cases and queries of this program compared to the other subject programs.

The results of the experiment indicate that overall, QRTTest outperformed all four control techniques for the remaining Java programs except for ( $T_4$ ). In two out of four Java programs (*JfreeChart* and *Xstream*) additional statement coverage technique ( $T_4$ ) performed better than QRTTest. In case of *JfreeChart*, code coverage based techniques ( $T_3$  and  $T_4$ ) performed better than other techniques. Looking into the characteristics of this program that we can see from Table I, in two versions of this program ( $P_{12}$ ) the number of code changes is very small (6 queries), which is the cause of the poor performance of text retrieval based techniques ( $T_5$ ,  $T_6$ ). Further analysis of the queries in *XStream* that degraded the performance revealed that, in most cases, the length of the original queries in this system was long ( $\sim > 150$  words), where not only query reformulation would not be effective but also the standard IR-based technique performs poorly. In such cases, the retrieval algorithm returns a tremendous number of

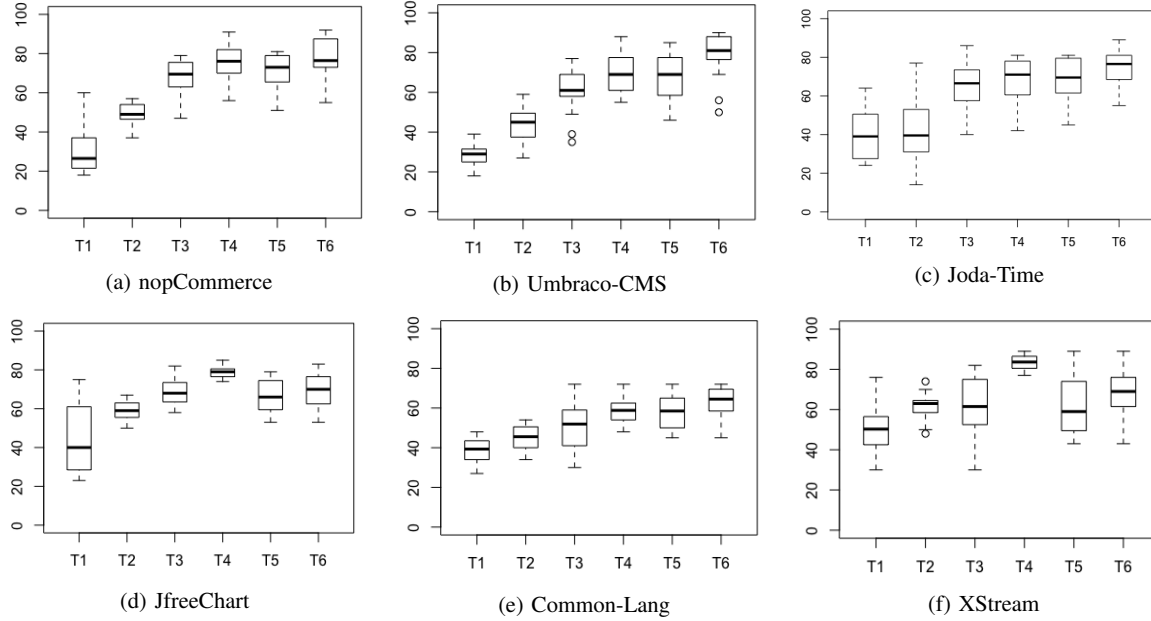


Figure 3: APFD box plots, all programs, all techniques. The horizontal axes list techniques, and the vertical axes denote APFD scores.

documents that contain some of the query terms, however, so many of those documents are not closely relevant to the query. This is because many of the query terms are frequently repeated in the majority of the source files. In such cases, query reduction is needed to tailor the query in a way that preserves the most significant terms and cut off the remaining terms.

## V. DISCUSSION

Our results indicate that the effectiveness of test prioritization can be improved through the use of query reformulation. In this section, we discuss additional observations and implications of our results and address the limitations of our work.

### A. Parameter Sensitivity Analysis

As shown in Figure 3, the use of QRTTest produced better APFD values compared to other testing techniques. However, in the query extension phase, we set the interpolation parameter ( $\lambda$ ) to a constant value and we also limited the number of expansion terms to 10. In this section, we want to examine how different values of  $\lambda$  and number of expansion terms can impact the accuracy of the retrieval. To investigate the impact of the parameter setting we run the experiment by assigning different values to  $\lambda$ , while growing the range of expansion terms by 5 times. The parameter  $\lambda$  determines the weight to be accorded to the new terms to be added to a query in relation to the original terms. Consequently, the number of terms to be added to the original query will increase the retrieval ratio.

Figure 4 shows the impact of these parameters on the retrieval performance. With regard to  $\lambda$ , as can be seen in Figure 4, the precision reaches its peak point for  $\lambda = 0.8$  for *nopCommerce* and *XStream*. And in all subject program

except for *Common-Lang*, retrieval technique achieved better accuracy for higher values of  $\lambda$ . In case of number of selected terms for expansion, we set 10 different ranges of term expansion for the retrieval from 5 to 50. Fixing these parameters at certain chosen values empirically is a common approach for evaluating the query reformulation methods [13]. From this figure, we can see that the retrieval technique achieved better accuracies for lower values and it remains steady as the number of term expansions exceeded 25. Almost in all subject programs the accuracy was noticeably higher when the number of selected terms were between 10 to 20.

### B. Implications of QRTTest

As we mentioned earlier, modern software development tend to grow much faster than before, however, often projects' budgets 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 [22]). QRTTest addresses this limitation by providing a powerful and yet novel technique that can reduce the cost of regression testing by prioritizing test cases based on the term similarity measure. QRTTest is an IR-based technique, which is efficient and flexible, and thus supports today's agile software delivery practices.

One major advantage of QRTTest is that its performance remains nearly constant even with the growth of the dataset. This aspect is particularly important for regression testing because the source code evolves 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).

While the application of the QRTTest leads to the improvement or preservation of the query performances in most cases, there are a couple of factors that impact its performance, which we discuss here. We discuss some examples and observations from the detailed analysis of the data. An example of a large improvement in query performance was observed on a query in the *nopCommerce* program. The initial queries of this program have two characteristics: short and non-discriminatory. When we applied query reformulation, after initial retrieval, QRTTest returned relevant documents that contained distinct terms, which lead to 57% improvement in recall and 61% increase in the overall precision compared to the standard IR technique.

Further, we investigate the correlation between QRTTest's performance and number of test cases, and the length of the queries. We calculate the Spearman Rank Correlation between the size of queries and APFD values, and between the number of test-methods and APFD values. The results show that there is a low correlation between the APFD value and the number of test cases ( $\rho = 0.16$ ), and there is a high reverse correlation between the performance of QRTTest and the length of the query ( $\rho = -0.67$ ). From these results, we can claim that the performance of QRTTest is independent from the number of test cases and is dependent on the length of the queries.

We also investigate whether the performance of QRTTest depends on the number of queries. We again calculate the Spearman Rank Correlation between the number of queries and APFD values. From our results, we also observe that the performance of QRTTest is independent from the number of queries ( $\rho = 0.24$ ). However, the performance of QRTTest drops significantly when the numbers of queries are noticeably 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).

### C. Cost-Benefit Analysis

As mentioned earlier one major goal of this study is to reduce the cost overhead of RTP. We showed that by applying IR-based technique we could eliminate the cost of the code coverage profiling. 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 the given situations, we discuss the cost-benefits of QRTTest and other control techniques. To perform the cost-benefit analysis, we divided the costs for applying QRTTest into three categories: 1) initial cost, 2) incremental cost, and 3) execution cost. The initial cost in this study is the cost of data construction (e.g., documents construction and setting properties values), and the cost of tool implementation.

The incremental cost in this study includes updating the document database (adding new test cases) and building new queries. This cost varies on the frequency of code changes. This cost for most applications is negligible. On average, updating the dataset for all programs took 4 seconds. There are four major elements that effect the execution cost of applying QRTTest: number of source code tokens, number of test cases, length of queries and number of selected terms for query

expansion. Number of source code tokens, query length and selected terms will impact the indexing time of QRTTest. On average, it took only 7.67 seconds for QRTTest to reorder the test cases. In the worst case scenario, for *Umbraco-CMS*, the application with the highest number of tokens and test cases, it took 11.28 seconds to perioritize test cases, under a term expansion limit of 10.

In case of control techniques, *Untreated* and *Random* do not have any cost overhead since they execute test cases based on the original order and randomly, respectively. The only cost of these techniques is test execution, although, our results show that these techniques are least effective among control techniques. The main costs of two code coverage-based techniques ( $T_3$  and  $T_4$ ) 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 prioritization using the additional code coverage technique is 40.66 minutes (the number of test cases in *nopCommerce v2.10* are smaller than other projects). This cost includes 28.35 minutes for profiling, 4.66 minutes for test execution and 9.13 seconds for calculation (sorting tests based on their additional code coverage). However, using QRTTest for the same program, the total prioritization cost is less than five seconds (4.48). Thus, using QRTTest reduced the cost up to 99.98% compared to coverage-based techniques.

Considering the cost-benefit tradeoffs among techniques that we discussed so far, we believe that QRTTest would offer great savings in general. Further, if we apply QRTTest to large scale industrial applications in which test execution might take several days (e.g., *mysql*, *Chrome*, etc.) [23], QRTTest will provide even greater savings. We conclude that the query reformulation recommendations formulated by QRTTest leads to the improvement of regression test case prrioritization performance in terms of both increased accuracy and fault detection capability.

## VI. THREATS TO VALIDITY

The main threat to validity is the technique we used to calculate the term similarity. As we explained in the approach section we applied cosine similarity formula to calculate the similarity between the code changes and test cases. Therefore, this technique does not apply to different types of testing such as GUI test etc. The choice of query reformulation algorithm impacts the internal validity of our results. In this research, we applied Relevance Model reformulation techniques, however, we do not know how would the results be affected if we use other measures or reformulation techniques. Another threat to internal validity is the values that we assigned to the number of expansion terms selected and the interpolation parameter during the query extension. We extended the queries by 10 words and we assigned  $\lambda = 0.8$ , however, using different values might produce different results. We also do not know how the results would change if we increased the size of the training datasets. The external validity refers to the

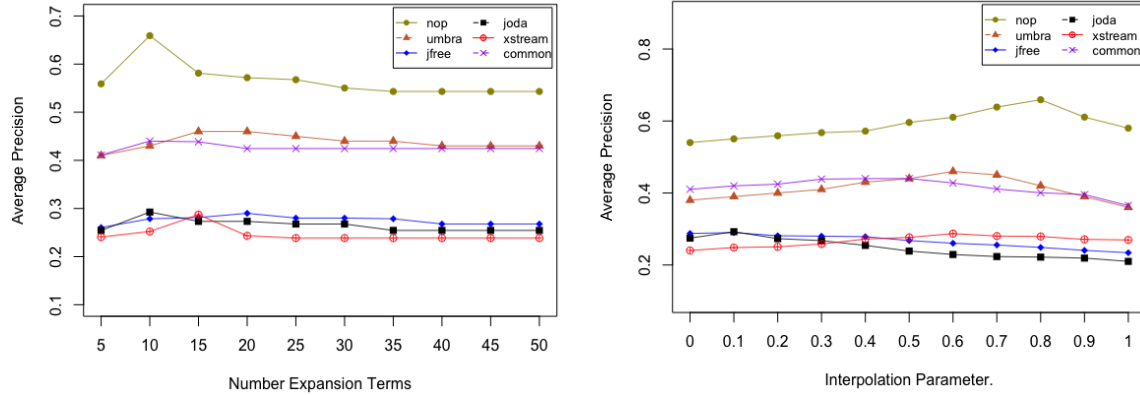


Figure 4: The Effects of Varying Number of Expansion Terms and Interpolation Values.

generalization of our findings. In order to address this threat, we selected six open source software systems from diverse domains, implemented in two programming languages, Java and C#. A larger set of code commits and larger systems, i.e., industrial applications, would strengthen the results from this perspective.

## VII. RELATED WORK

For more than two decades, reducing the time and cost of regression testing has been an active research topic. Recent surveys on regression testing techniques [11], [40] provide a comprehensive understanding of overall trends of the techniques and areas for improvement. To date, numerous regression testing techniques have been implemented using various types of data sources (e.g., code coverage, test case diversity, and fault history). The code coverage-based technique is one of the most widely used and evaluated techniques [16], [17], [20], [33], [8], [9]. Despite the effectiveness of the aforementioned approaches, these techniques suffer from the computational overhead and the demand for collecting and analyzing different test quality metrics (e.g. code coverage, fault history, etc.), which makes them less practical for modern software development environment. Considering the increasing demands on rapid development and release cycles, techniques that require expensive application costs would not be suitable for modern software development practices.

In order to reduce the costs and overhead of regression testing techniques, some researchers have investigated automated techniques that can be aligned with modern software development environment. IR-based techniques are specific instances of these modern techniques. These techniques often receive a large size of collection of documents, and extract the most relevant documents based on the given query and ranking formula. Because these techniques are light-weight, efficient, and easy to implement, they have been applied to a variety of software engineering problems such as bug localization [15], traceability link recovery [27], [29], change impact analysis [12], clone detection etc [28].

More recently researchers have investigated the application of IR-based techniques in regression testing by reformulating

software testing problem as an information retrieval problem. In this approach, a code commit is used as a query to search for relevant code/test from source code or other software artifacts. The output of the system is a list of code documents that are relevant to the query. The relevance is measured by the textual similarity between the query and documents. [36], [10], [32], [34], [25], [38] are instances of this approach. In [36] authors proposed an IR-based 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 the cosine similarity to the queries. Similarly, Azizi and Do [10] proposed an IR-based technique for regression test selection. Their technique uses program changes, test suite source files, and fault history information to implement a network of test cases. The proposed system uses these sources of information to identifies a list of important test cases that are highly likely to detect regression defects. Unlike these previous attempts at improving IR-based technique in regression testing, our focus is on reformulating a low-quality query.

## VIII. CONCLUSIONS AND FUTURE WORK

In this research, we have introduced a novel technique, QRTest, to improve the effectiveness as well as reducing the cost of regression testing. QRTest uses the textual similarity of a program source code and returns a list of test cases that are highly likely covering the modified portion of the program. QRTest does not require any dynamic code coverage, static analysis, or user feedback, and also it is flexible, scalable, and language-independent. We evaluated our approach using 37 versions of 6 open-source software projects by comparing our approach to five other test prioritization techniques. Our empirical results indicate that the use of QRTest outperformed all baselines and leads to query performance improvement in 76% of the cases. For future work, we aim to investigate ways to further improve the proposed query reformulation technique. We also plan to investigate how this research applies to different areas of regression testing such as test case selection and reduction.



## REFERENCES

- [1] <https://www.devart.com/codecompare/>. [Accessed: Oct. 12, 2020].
- [2] <http://www.nopcommerce.com/>. [Accessed: Oct. 26, 2020].
- [3] <https://umbraco.com/>. [Accessed: Aug. 06, 2020].
- [4] <https://github.com/rjust/defects4j>. [Accessed: Oct. 25, 2020].
- [5] <https://github.com/nopSolutions/nopCommerce/issues/>. [Accessed: Oct. 06, 2020].
- [6] <http://issues.umbraco.org/issues/>. [Accessed: Oct. 06, 2020].
- [7] <http://www.eclEmma.org/jacoco/trunk/doc/maven.html/>. [Accessed: Aug. 06, 2020].
- [8] Maral Azizi and Hyunsook Do. A collaborative filtering recommender system for test case prioritization in web applications. In *Symposium On Applied Computing*, page 107–114. ACM, 2018.
- [9] Maral Azizi and Hyunsook Do. Graphite: A greedy graph-based technique for regression test case prioritization. In *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 245–251. IEEE, 2018.
- [10] Maral Azizi and Hyunsook Do. Retest: A cost effective test case selection technique for modern software development. In *International Symposium on Software Reliability Engineering*, pages 144–154. IEEE, 2018.
- [11] S. Biswas, R. Mall, M. Satpathy, , and S. Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011.
- [12] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *IEEE International Software Metrics Symposium (METRICS)*, pages 144–154. IEEE, 2005.
- [13] Claudio Carpineto and Giovanni Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Computing Surveys*, January 2012.
- [14] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 376–387. ACM, 2017.
- [15] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta and Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Foundations of Software Engineering*, pages 396–407. IEEE, 2017.
- [16] 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.
- [17] 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.
- [18] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proceedings of the International Conference on Software Maintenance*, pages 113–124, September 2005.
- [19] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, September 2006.
- [20] 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.
- [21] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *International Conference on Software Engineering (ICSE)*, pages 842–851. IEEE, 2013.
- [22] Hadi Hemmati, Zhihan Fang, and Mika V Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *International Conference on Software Testing, Verification, and Validation*, pages 19–26. IEEE, 2015.
- [23] 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.
- [24] J. Jenkins. Velocity culture (the unmet challenge in ops). In *Presentation at O'Reilly Velocity Conference*. IEEE, 2011.
- [25] Jung-Hyun Kwon, In-Young Ko, Gregg Rothermel, and Matt Staats. Test case prioritization based on information retrieval concepts. In *Asia-Pacific Software Engineering Conference*, pages 19–26. IEEE, 2014.
- [26] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 842–851. IEEE, 2015.
- [27] A. D. Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *International Conference of Software Maintenance*, page 299–309. IEEE, 2006.
- [28] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *International Conference of Automated Software Engineering*, page 107–114. IEEE, 2001.
- [29] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *International Conference of Software Engineering*, page 125–135. IEEE, 2003.
- [30] 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.
- [31] Laura Moreno, Gabriele Bavota, Sonia Haiduc, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, and Andrian 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.
- [32] 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*, pages 636–643. IEEE, 2011.
- [33] 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.
- [34] Simone Romano, Giuseppe Scanniello, Giuliano Antoniol, and Alessandro Marchetto. SPIRiTUS: a Simple Information Retrieval regression Test Selection approach. *Information and Software Technology*, July 2018.
- [35] G. Rothermel, R. Untch, C. Chu, , and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 179–188. IEEE-ACM, 1999.
- [36] 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.
- [37] Bunyamin Sisman and Avinash C. Kak. Assisting code search with automatic query reformulation for bug localization. In *Working Conference on Mining Software Repositories (MSR)*, pages 309–318. IEEE, 2013.
- [38] Sahar Tahvili, Leo Hatvani, Michael Felderer, Wasif Afzal, and Markus Bohlin. Automated functional dependency detection between test cases using doc2vec and clustering. In *International Conference on Artificial Intelligence Testing (AITest)*, pages 19–26. IEEE, 2019.
- [39] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic. In *Empirical Software Engineering*, pages 1–31. IEEE-ACM, 2012.
- [40] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.