REPiR: An Information Retrieval based Approach for Regression Test Prioritization

Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, Dewayne E. Perry
Electrical and Computer Engineering, The University of Texas at Austin, TX 78712, USA
{ripon, zhanglm}@utexas.edu, {khurshid, perry}@ece.utexas.edu

ABSTRACT

Regression testing is widely used in practice for validating program changes. However, running large regression suites can be costly. Researchers have developed several techniques for *prioritizing* tests such that the higher priority tests have a higher likelihood of finding bugs. A vast majority of these techniques are based on *dynamic* analysis, which can be precise but can also have significant overhead (e.g., for program instrumentation and test-coverage collection). We introduce a new approach, REPiR, to address the problem of regression test prioritization by reducing it to a standard Information Retrieval problem such that the differences between two program versions form the *query* and the tests constitute the *document collection*. REPiR does not require any dynamic profiling or static program analysis. As an enabling technology we leverage the open-source IR toolkit Indri. An empirical evaluation using seven open-source Java projects shows that REPiR is computationally efficient and performs significantly better than many existing (dynamic or static) techniques while performs at least as well for others.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: [Testing and Debugging]; H.4 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation, Measurement

Keywords

Regression Testing, Test Prioritization, Information Retrieval

1. INTRODUCTION

Programs commonly evolve due to feature enhancements, program improvements, or bug fixes. Regression testing is a widely used methodology for validating program changes. However, regression testing can be time consuming and expensive [3, 24]. Executing a single regression suite can even take weeks [40]. Therefore, early detection of regression faults is highly desirable.

Regression test prioritization (RTP) is a widely studied technique that ranks the tests based on their likelihood in revealing faults and defines a test execution order based on this ranking so that tests that are more likely to find (new, unknown) faults are run earlier [12, 34, 50, 53, 55]. Existing RTP techniques are largely based on dynamic code coverage where the coverage from the previous program version is used to order, i.e., *rank*, the tests for running against the next version [12, 50, 53, 21]. A few recent techniques utilize static program analysis in lieu of dynamic code coverage [34, 55]. RTP techniques (whether dynamic or static) are broadly classified into two categories, *total* or *additional*, depending on how they calculate the rank [40]. *Total* techniques do not change values of test cases during the prioritization process, whereas *additional* techniques adjust values of the remaining test cases taking into account the influence of already prioritized test cases.

Even though a number of existing RTP techniques (specifically coverage-based ones) have been widely used, they have two key limitations [34]. First, coverage profiling overhead (in terms of time and space) can be significant. Second, in the context of certain program changes (which modify behavior significantly) the coverage information from the previous version can be imprecise to guide test prioritization for the current version. Although the static techniques [55, 34] address the coverage profiling overhead, they simulate the coverage information via static analysis, and thus can be also imprecise.

This paper presents REPiR, which introduces a new dimension of techniques for regression test prioritization based on information retrieval (IR) [30]. Traditional IR techniques focus on the analysis of natural language in an effort to find the most relevant *documents* in a collection based on a given *query*. An IR system generally provides a ranked-list of documents based on the relevance between the given documents and query. Even though the original focus of IR techniques was on documents written in natural language, recent years have seen a growing number of applications of IR to effectively solve software engineering problems by extracting useful information from source code and other software artifacts [26, 33, 36, 7, 28]. The effectiveness of these solutions relies on the use of meaningful terms (e.g., identifiers and comments) in software artifacts, and such use is common in projects written in languages, such as Java and C#, which encourage it among the best practices.

Our key insight is that in addition to writing good iden-

FSE '14, Hongkong

tifier names and comments in the code, developers use very similar terms for test cases, and we can utilize these textual relationships by reducing the RTP problem to a standard IR problem such that program changes constitute the query and the test cases form the document collection. Thus, we define a natural connection between RTP and IR. Our tool REPiR embodies our insight. We build REPiR on top of the state-of-the-art Indri [46] toolkit, which provides an open-source, highly optimized platform for building solutions based on IR principles. Thus, REPiR provides an efficient approach to addressing the RTP problem without requiring any dynamic coverage information or static program analysis.

We evaluate REPiR using a dataset consisting of seven open-source software projects with real regression faults. Specifically, we compare REPiR against ten strategies for RTP, including four *total* and four *additional* strategies. The experimental results show that REPiR significantly outperforms each *total* program-analysis or coverage-based strategies. REPiR also overall outperforms *additional* strategies although the mean difference is not statistically significant. We also experimented with different variants of REPiR and provide detailed results how REPiR can be used more effectively depending on test or program differencing granularities. The results show that REPiR is slightly more effective with high-level program differences at method-level tests and low-level differences at class-level tests. For reproducibility and verification, our experimental data is available online.[1] We make the following contributions:

- **RTP using IR**. We introduce the idea of using information retrieval (IR) for regression test prioritization (RTP). To our knowledge, previous work has not applied IR to RTP.
- **REPiR**. We define a reduction from the regression test prioritization problem to a standard information retrieval problem and present our approach, REPiR, based on this reduction.
- **Tool**. We embody our approach into a prototype tool that leverages off-the-shelf, state-of-the-art Indri toolkit for information retrieval.
- **Evaluation**. We present a rigorous empirical evaluation using seven open-source Java projects with real regression faults and compare REPiR with 10 RTP strategies. The results show that REPiR is computationally efficient and performs significantly better than all *total* (dynamic or static) strategies while matching the accuracy of all *additional* strategies.

2. BACKGROUND

This section briefly discusses the basic concepts of regression test prioritization, working procedure of an IR system, and its applications in software engineering.

2.1 Regression Test Prioritization

A test prioritization technique or tool reorders the actual execution sequences of test cases in such a way that it meets certain objectives of developers or testers. The nature of objectives could be diverse including, but not limited to, increasing the overall fault detection rate or increasing code coverage at a faster rate.

[1]https://www.dropbox.com/s/rsg1yk5x65bjh2u/repir.zip

Rothermel et al. [40] formally defined the test case prioritization problem as finding $T' \in PT$, such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$. In the definition, PT denotes the set of all possible permutations of a given test suite T, and f denotes a function from any member of PT to a real number such that a larger number indicates better prioritization. In this paper, we focus on increasing the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process.

2.2 IR Techniques in Software Engineering

Recent years have seen many IR applications applied to solve software engineering problems. IR is concerned with search: given a massive collection of documents, IR seeks to find the most relevant documents based on a user-query. Generally an IR system works in three major steps: text preprocessing, indexing, and retrieval. Preprocessing involves text normalization, stop-word removal, and stemming. A text normalizer removes punctuation, performs case-folding, tokenizes terms, etc. In the stop-word removal phase, an IR system filters out the frequently used terms such as prepositions, articles, etc. in order to improve efficiency and reduce spurious matches. Finally, stemming conflates variants of the same term (e.g., go, going, gone) to improve term matching between query and document. Then the documents are indexed for fast retrieval. Once indexed, queries are submitted to the search engine, which returns a ranked-list of documents in response. Finally, the search engine (or "model") is evaluated by measuring the quality of its output ranked-list relative to each user's input query. For a broad overview of IR, please refer [30] online.

IR techniques have been applied to over two dozens of different software engineering problems, many of which are highlighted in two surveys on the application of IR to SE problems [5, 6]. There are two predominant tasks today. One is feature (or concept) location, which consists of locating features described in maintenance requests, such as enhancements or faults [26, 33, 36, 16, 7, 28]. The second task is traceability, which links or recovers links between software engineering artifacts [27, 32]. Another closely related task is software reuse, where IR is used to identify the reusable software artifacts [15]. There are also a diverse set of other tasks such as quality assessment [22], change impact analysis in source code [8], restructuring and refactoring [2], defect prediction [4], clone detection [31] and duplicate bug detection [47].

3. REPiR APPROACH

Regression test prioritization (RTP) and Information Retrieval (IR) both deal with a *ranking* problem, albeit in different domains (Section 2). While RTP is concerned with test cases written in programming language, IR is concerned with documents written in natural language. However, many human-centric software engineering documents are text-based, including source-code, test scripts, and test documents. Furthermore, in real world software projects, developers often use meaningful identifier names and write comments, which allow solving a number of software engineering problems using information retrieval. Our key insight is that in addition to writing good identifier names and comments in the code, developers use very similar terms for test cases, and we can utilize these textual relationships using IR to develop efficient RTP techniques.

3.1 Problem Formulation

We reduce RTP as a standard IR problem where the program difference between two software revisions or versions is the *query* and the test cases or test-classes are the *document collection*. Therefore, for a given test suite TS, the prioritized test suite TS' is defined by the ranking of tests in TS based on the similarity score between the program difference and test cases. Reducing the RTP technique to a standard IR task enables us to exploit a wealth of prior theoretical and empirical IR methodology, providing a robust foundation for tackling RTP. In this work, we primarily focus on projects with JUnit test-cases.

3.2 Construction of Document Collection

The process of constructing documents from test cases varies depending on the information granularity and the choice of information retrieval techniques. Generally a test suite is a collection of source code files, where each source code file consists of one or more test-methods/functions. For example, JUnit has two levels of test cases: test-classes and test-methods. Prior researches [55, 53] on RTP focused on prioritizing both test-methods and test-classes. In this section, we describe the construction of three types of document collections. Hereafter, we use test-class to denote test-class/file and test-method to denote test-method/function.

3.2.1 At Test-Class Level

To make document collection at test-class level, we first build the abstract syntax tree (AST) of each source code file using Eclipse Java Development Tools (JDT), and traverse the AST to extract all the identifier names (class names, method names, and variable names) and comments. The identifier names and comments are particularly important from the information retrieval point of view, since these are the places where developers can use their own natural language terms. Note that a document collection at the test-class level can be also constructed without any knowledge of underlying programming language. In this case, we do not construct any AST for test-classes. Rather, we read each term in the test class, remove all mathematical operators using simple text processing, and tokenize them to construct the document-collection.

3.2.2 At Test-Method Level

For constructing document collection at the test-method level, we extract all the methods from the AST using JDT and store all the identifiers and comments related to a given method as a text document.

3.2.3 Structured Document

For structured information retrieval, it is important to store documents in such a way that they retain the program structure. In our study, we distinguish four kinds of terms based on program constructs: i) class names, ii) method names, iii) all other identifier names such as variable names, api names, and iv) comments. To construct structured documents, we traverse the AST for either each test-class or test-method to extract aforementioned terms and store them in an XML document.

3.3 Query Construction

As we defined in the problem formulation, in an IR-based RTP, differences between two program versions comprise the query. How to utilize the best query representation (e.g., succinct vs. descriptive)–is a very well-known problem in traditional IR [23]. While the succinct representation often provides the most important keywords, it may lack other terms useful for matching. In contrast, although the more verbose descriptions may contain many other useful terms to match, it may also contain a variety of distracting terms. In our work, we experiment with three representations of program differences that can affect the overall results of RTP.

Table 1: High Level Change Types

No.	Type	Change Description
1	CM	Change in Method
2	AM	Addition of Method
3	DM	Deletion of Method
4	AF	Addition of Field
5	DF	Deletion of Field
6	CFI	Change in Instance of Field Intializer
7	CSFI	Change in Static Field Initializer
8	LC_m	Look-up Change due to Method Changes
9	LC_f	Look-up Change due to Field Changes

3.3.1 Low Level Differences

By low level differences, we mean the program differences between two versions at the line level. We compute the low level differences by applying UNIX *diff* recursively while ignoring spaces and blank lines. It can be also obtained from program versioning system (e.g. cvs, svn, git) without any computation. We denote this representation of query as `LDiff` and quantify it in terms of number of lines.

3.3.2 High level differences

Since LDiff is expected to be noisy (e.g., from changes in formatting, or local changes), our goal is to summarize LDiff by abstracting local changes and ignoring formatting differences. To this end, we consider nine types of atomic changes (Table 1) that have been used in various studies for change impact analysis. We use FaultTracer [54], a change impact analysis tool, to extract these changes. We denote this high level query as `HDiff` and quantify it in terms of number of the atomic changes.

3.3.3 Compact LDiff or HDiff

Since difference between two program versions is often too long (e.g. thousands of lines), it is highly likely that they would have many duplicated terms. In this representation, we construct a compact version of LDiff and HDiff by removing all the duplicated words from them. We denote these compact forms of LDiff and HDiff as `LDiff.Distinct` and `HDiff.Distinct` respectively.

3.4 Tokenization

Since we are dealing with program source code rather than natural language written in English, similar to other IR systems for software engineering, our tokenization is different than that of standard IR task. Generally identifier names are often a concatenation of words. Dit et al. [8] compared simple camel case splitting to the more sophisticated Samurai [10] system and found that both performed comparably in concept location. Therefore, in addition to splitting terms based on period, comma, white space, we also split identifier

names based on the camel case heuristic.

3.5 Retrieval Model

Researchers in software engineering have experimented with a number of different retrieval models developed by IR including latent semantic indexing (LSI) [9], the vector space model (VSM) [30], and Latent Dirichlet Allocation (LDA) [49]. However, recent research shows that TF.IDF term weighted VSM (briefly TF.IDF model) works better than others [41, 56]. Another study shows that although there is a widespread debate on which of three (TF.IDF [42], BM25 (Okapi) [37], or language modeling [35]) traditionally-dominant IR paradigms was best, all three approaches utilize the same underlying textual features, and empirically perform comparably when well-tuned [14]. Therefore, we chose the TF.IDF model for our study. We elaborate on this TF.IDF formulation below.

Let us assume that test cases (documents) and a program difference (query) are represented by a weighted term frequency vector $\vec{d}$ and $\vec{q}$ respectively of length n (the size of vocabulary, i.e., the total number of terms).

$$\vec{d} = (x_1, x_2, ..., x_n) \quad (1)$$

$$\vec{q} = (y_1, y_2, ..., y_n) \quad (2)$$

Each element of x_i in $\vec{d}$ represents the frequency of term t_i in document d (similarly, y_i in query $\vec{q}$). However, the terms that occur very frequently in most of the documents are less useful for search. Therefore, in a vector space model, generally query and document terms are weighted by a heuristic TF.IDF weighting formula instead of only their raw frequencies. Inverse document frequency (IDF) diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. Thus, weighted vectors for $\vec{d}$ and $\vec{q}$ are:

$$\vec{d_w} = (tf_d(x_1)idf(t_1), tf_d(x_2)idf(t_2), ..., tf_d(x_n)idf(t_n)) \quad (3)$$

$$\vec{q_w} = (tf_q(y_1)idf(t_1), tf_q(y_2)idf(t_2), ..., tf_q(y_n)idf(t_n)) \quad (4)$$

The basic formulation of IDF for term t_i is $idf(t_i) = log\frac{N}{n_{t_i}}$, where N is the total number of documents in C and n_{t_i} is the number of documents with term t_i. Therefore, in the simplest TF.IDF model, we would simply multiply this value by the term's frequency in document d to compute the TF.IDF score for (t, d). However, actual TF.IDF models used in practice differ greatly from this to improve accuracy [44, 37]. To date IR researchers have proposed a number of variants of TF.IDF model. We adopt Indri's built-in TF.IDF formulation, based upon the well-established BM25 model [37, 52]. This TF.IDF variant has been actively used in IR community over a decade and rigorously evaluated in shared task evaluations at the Text REtrieval Conference (TREC). In this variant, the *document*'s tf function is computed by Okapi:

$$tf_d(x) = \frac{k_1 x}{x + k_1(1 - b + b\frac{l_d}{l_C})} \quad (5)$$

where k_1 is a tuning parameter (≥ 0) that calibrates document term frequency scaling. The *term frequency* value quickly saturates for a small value of k_1, whereas, a large value corresponds to using raw term frequency. b is another tuning parameter between 0 and 1, which is the document scaling factor. When the value of b is 1, the term weight is fully scaled by the document length. For a zero value of b, no length normalization is applied. l_d and l_C represents the document length and average document length for the collection respectively.

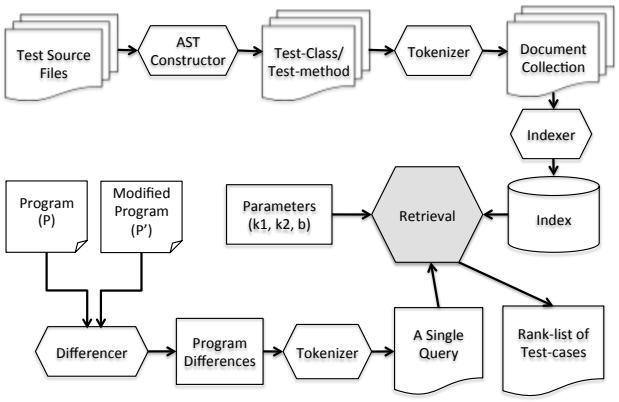

Figure 1: REPiR Architecture

The IDF value is smoothed as $log\frac{N+1}{n_t+0.5}$ to avoid division by zero for the special case when a particular term appears in all documents.

The term frequency function of query, tf_q is defined similarly as tf_d. However, since the query is fixed across documents, normalization of query length is unnecessary. Therefore, b is set simply to zero.

$$tf_q(y) = \frac{k_2 y}{y + k_2} \quad (6)$$

Now the similarity score of document $\vec{d}$ against query $\vec{q}$ is given by Equation 7.

$$s(\vec{d}, \vec{q}) = \sum_{i=1}^{n} tf_d(x_i) tf_q(y_i) idf(t_i)^2 \quad (7)$$

3.6 Structured Information Retrieval

The TF.IDF model presented in Equation 7 does not consider source code structure (program constructs)–i.e., each term in a source code file is considered having the same relevance with respect to the given query. In our recent work on IR-based bug localization [41], we found that incorporating structural information into IR model, known as structured retrieval, can help improve the results considerably. In this paper, we adapt the structured IR for RTP.

As we described in Section 3.2.3, we distinguish four different program constructs: *class, method, variables, comments*. Therefore, to exploit all of this structural information, we perform a separate search for each type of terms. Since TF.IDF model assumes that there is no statistical dependence between terms, we simply sum all the field retrieval scores to calculate the final similarity score.

$$s'(\vec{d}, \vec{q}) = \sum_{f \in D} s(d_f, q) \quad (8)$$

where f is a particular document field. The benefit of this model is that terms appearing in multiple document fields are implicitly assigned greater weight, since the contribution from each term is summed over all fields in which it appears.

3.7 Architecture

Figure 1 shows the overall architecture of our IR-based RTP prototype, REPiR (REgression test Prioritization using information Retrieval). First, REPiR takes the source

code files of tests as input that we would like to prioritize. Next, it extracts information from test cases, tokenizes the terms, and constructs a document collection for a given level of granularity as described in Section 3.2. REPiR also extracts program changes (LDiff, HDiff, or compact) and tokenizes terms in the same way as tokenizing document terms to construct the query.

We adopt the Indri toolkit [46], a highly efficient open-source library, for indexing and developing our retrieval model. After documents and queries are created above, they are handed off to Indri for stopword removal, stemming, and indexing. Note that we use the default stopword list provided with Indri and Krovetz stemmer for stemming. We also set the values of k_1, k_2, and b to 1.0, 1000, and 0.3 respectively, which have been found to perform well empirically [41].

4. EMPIRICAL EVALUATION

To investigate the effectiveness of REPiR, we performed an empirical evaluation in terms of five research questions.

- **RQ1:** Is REPiR more effective than random or untreated test case orders?

  This research question aims to understand whether REPiR reveals regression faults earlier than when there is no RTP or test cases are ordered at random.

- **RQ2:** Do the high-level program differences help improve the performance of REPiR?

  Low-level program differences are expected to produce noisy results since changes to even a single character of a line would be interpreted as deletion of a line followed by an addition of line. Therefore, in this research question, we investigate whether the high-level program differences based on AST help improve the accuracy of REPiR.

- **RQ3:** Is structured retrieval more effective for RTP?

  In our recent work, we showed that incorporating the structural information program into traditional information retrieval model, which is known as structured information retrieval, can improve the bug localization results considerably [41]. In this research question, we investigate whether the same is true for IR-based RTP.

- **RQ4:** How does REPiR perform compared to the existing RTP techniques?

  To date researchers have focused on various program analysis (either static or dynamic) based techniques to propose or improve RTP. In this research question, we are interested in investigating how well REPiR performs compared to those existing techniques.

- **RQ5:** How does REPiR perform when it is oblivious to language-level information?

  Since REPiR only utilizes textual information, identification of specific programing language constructs, such as control-flow structures, is not needed. Lightweight languange-specific parsing is used only for identifier-name extraction for constructing document collection at the method-level. However, if we use LDiff to prioritize test-classes, REPiR can be made completely oblivious of the underlying programming language. In this research question, we investigate how REPiR performs for this configuration.

4.1 Subject Systems

We studied seven open source software systems for our study. All of these systems are from diverse application domain and have been widely used in software testing research [11, 34, 43]. We obtained Xml-Security from the well-known Software-artifact Infrastructure Repository (SIR) [10] and extract other subject systems from their host website. The sizes of these systems vary from 5.7K LOC (Time and Money, 2.7K LOC source code and 3K LOC test code) to 88.8K LOC (Joda-Time, 32.9K LOC source code and 55.9K LOC test code).

For each subject systems, first we extract all the major releases with their test cases and consider each consecutive versions as a version-pair. For each pair, we run the old regression test suite on new version to see if there is any regression fault. In this way, we were able to identify 20 version-pairs with regression faults, which we used in our study. Table 2 provides all the details regarding each version-pair including the number of methods and classes, the number of test-classes and test-methods, the size of program edits, the number of faulty edits, and fault-revealing test cases.

4.2 Independent Variable

Since we are interested in investigating the performance of REPiR for different granularities of test cases, different representations of program differences, and how it work compared to other RTP strategies, we have mainly three independent variables:

- Test-case Granularity
- Program Differences, and
- Prioritization Strategy

In section 3, we discussed different test cases granularities and program differences. Now we briefly describe 10 test prioritization strategies that we considered for comparison.

Untreated test prioritization keeps the original sequence of test cases as provided by developers. In our discussion, we denote the untreated test case prioritization as `UT`. We consider this to be the *control* treatment.

Random test prioritization rearranges test cases randomly. Since the results of random strategy may vary a lot for each run, we applied random test prioritization 20 times for each subject and considered the average as final result. In our discussion, we denote the random test prioritization technique as `RT`.

Dynamic coverage-based test prioritization varies depending on the types of coverage information (e.g., the method or statement coverage) and prioritization strategies (e.g., the *total* or *additional* strategy). We used the four most-widely used variants of coverage-based RTP: `CMA`, `CMT`, `CSA`, and `CST`. For example, `CMA` denotes test prioritization based on the *method* coverage using the *additional* strategy, and `CST` denotes test prioritization based on the *statement* coverage using the *total* strategy.

JUPTA [55, 34] is a static program analysis based test prioritization approach that sorts the test cases based on test ability (`TA`). TA is determined by the number of program elements relevant to a given test case (`T`), which is computed from the static call graph of T to simulate coverage information. TA can be calculated based on two levels of granularity: fine granularity and coarse granularity. TA at the

Table 2: Description of Dataset

No.	Project	Version Pair	#TMethods	#TClass	#FTMethods	#LDiff	#HDiff	#FEdits
P_1	Time and Money	3.0-4.0	143	15	1	1200	215	1
P_2	Time and Money	4.0-5.0	159	16	1	658	246	1
P_3	Mime4J	0.50-0.60	120	24	8	4377	2862	3
P_4	Mime4J	0.61-0.68	348	57	3	2967	3160	4
P_5	Jaxen	1.0b7-1.0b9	24	12	2	2788	204	3
P_6	Jaxen	1.1b6-1.1b7	243	41	2	5688	473	5
P_7	Jaxen	1.0b9-1.0b11	645	69	1	1020	92	1
P_8	Xml-Security	1.0-1.1	91	15	5	6025	329	2
P_9	XStream	1.20-1.21	637	115	1	833	209	1
P_{10}	XStream	1.21-1.22	698	124	2	1079	222	2
P_{11}	XStream	1.22-1.30	768	133	11	5920	540	11
P_{12}	XStream	1.30-1.31	885	150	3	2630	416	3
P_{13}	XStream	1.31-1.40	924	140	7	6744	1225	7
P_{14}	XStream	1.41-1.42	1200	157	5	828	136	5
P_{15}	Commons-Lang	3.02-3.03	1698	83	1	1757	221	1
P_{16}	Commons-Lang	3.03-3.04	1703	83	2	3003	172	2
P_{17}	Joda-Time	0.90-0.95	219	10	2	8653	5976	2
P_{18}	Joda-Time	0.98-0.99	1932	71	2	13735	1254	2
P_{19}	Joda-Time	1.10-1.20	2420	90	1	1348	793	1
P_{20}	Joda-Time	1.20-1.30	2516	93	3	1979	571	3

fine-granularity level is calculated based on the statements contained by the methods called by a test case, whereas TA at the coarse-granularity level is calculated based on the methods called by a test case. Similar with coverage-based prioritization techniques, we also used four variants of JUPTA: `JMA`, `JMT`, `JSA`, and `JST`.

Note that we implemented all the static and dynamic RTP techniques using byte-code analysis. More specifically, we used the *ASM byte-code manipulation framework*[2] to extract all the static and coverage information for test prioritization.

4.3 Dependent Variable

We use the Average Percentage Faults Detected (APFD) [40], a widely used metric in evaluating regression test prioritization techniques, as the dependent variable. This metric measures prioritization effectiveness in terms of the rate of fault detection of a test suite, and is defined by the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2 \times n} \quad (9)$$

where n denotes the total number of test cases, m denotes the total number of faults, and TF_i denotes the smallest number of test cases in sequence that need to be run in order to expose the i^{th} fault. The value of APFD can vary from 0 to 1. Since n and m are fixed for any given test suite, higher APFD values indicate higher fault-detection rates.

4.4 Study Results

In this section, we present the experimental results which answer our research questions.

4.4.1 Performance: REPiR vs. UT and RT

First, to understand the performance of REPiR compared to UT and RT at the test-method level, REPiR is set to construct the document collection at test-method level and use LDiff as a query. We select the TF.IDF model as the underlying retrieval technique and run REPiR on all version pairs (P_1-P_{20}). We also run RT and UT for the same dataset. Each RTP technique provides a ranked-list of test-methods for each version-pair, which we use to calculate APFDs. We also perform the same experiment for test-class level.

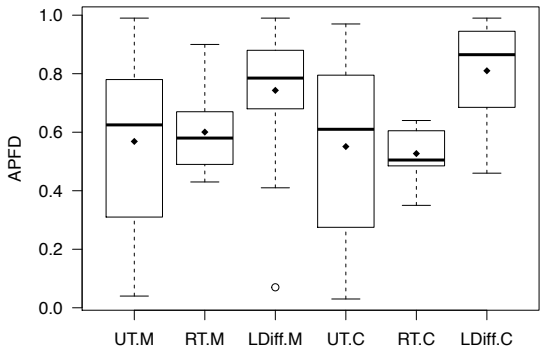

Figure 2: Performance of REPiR(LDiff) at Test-Method and Test-Class Level

Figure 2 presents the results for all version-pairs in form of boxplot. In each plot, the X-axis shows the strategies that we compared and the Y-axis shows the APFD values. To name RTP techniques, we used M to denote method-level and C to denote class-level test-cases. Each boxplot shows the average (dot in the box), median (line in the box), upper/lower quartile, and 90th/10th percentile APFD values achieved by a strategy. From the figure, we see that the mean, median, first and third quartiles APFD of (UT, RT, REPiR) at test-method level are (0.57, 0.6, **0.74**), (0.63, 0.58, **0.79**), (0.35, 0.49, **0.69**), and (0.75, 0.67, **0.87**) respectively, which clearly indicates that REPiR overall performs better than UT and RT.

We also investigate whether the accuracy of REPiR vary with the length of program differences or the number of test-methods, since these are two main inputs for REPiR. We compute the Spearman correlation between the size of LDiff

[2]http://asm.ow2.org/

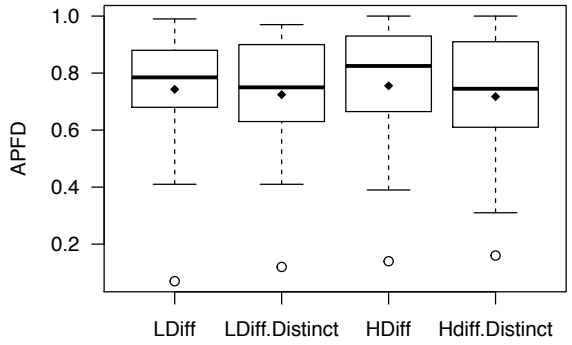

Figure 3: Impact of Program Differences at test-method Level

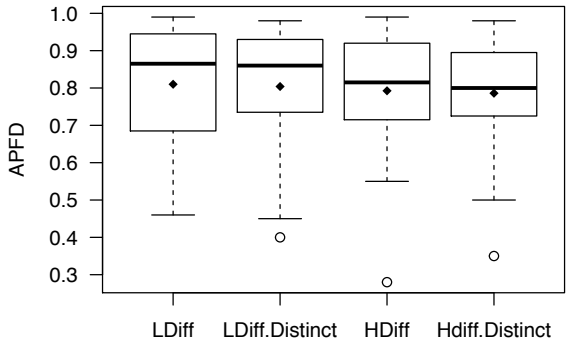

Figure 4: Impact of Program Differences at test-class Level

(quantified by number of changed lines) and APFD, and between the number of test-methods and APFD. The low correlation values for both cases (0.19 and 0.44) indicate that the accuracy of REPiR is fairly independent of the length of program differences and the number of test-methods.

Similarly for the test-class level, we see that the mean, median, first and third quartiles APFD of REPiR (**0.81**, **0.87**, **0.7**, **0.94**) is higher than that of UT (0.55, 0.61, 0.3, 0.76) and RT (0.53, 0.51, 0.49, 0.6). These results show that REPiR performs much better than UT and RT at test-class level. The low Spearman correlations between the number of test-classes and APFD (0.23) and between the length of program differences and APFD (-0.4) indicate that the accuracy of REPiR is not dependent either on the the number of test-classes and size of program differences.

4.4.2 Impact of Program Differences

To answer RQ2, we run REPiR with four forms of program differences (LDiff, LDiff.Distinct, HDiff, HDiff.Distict) separately for both at test-method level and at test-class level.

Figures 3 and 4 present the summary of APFD values for test-methods and test-classes respectively. From Figure 3 we see that at method-level HDiff works better than the LDiff in terms of both mean (HDiff: 0.76 vs. LDiff: 0.74) and median (HDiff: 0.83 vs. LDiff: 0.79). When we take a closer look into our data for individual program versions, we find that HDiff improves the APFD values for 12 version-pairs, while decreases it for 5 version-pairs. However, if we further condense the query by removing duplicate terms, the accuracy decreases for both HDiff and LDiff and the drop is higher for HDiff than that of LDiff. We believe that since HDiff is already condensed, it was affected more due to the removal of duplicated terms than LDiff.

On the other hand, From Figure 4, we see that LDiff works better than HDiff at test-class level. The mean and median APFDs for HDiff are 0.79 and 0.82 respectively, whereas they are 0.81 and 0.87 for LDiff. From the results of LDiff.Distinct and HDiff.Distinct, we see that like test-method level, the removal of duplicated terms hurt the results as well. However, at test-class level the drop is very small compared to that of test-method level.

4.4.3 Impact of Structured Retrieval

To understand whether the structured retrieval leads to better prioritization, we constructed the structured version of document collection at both test-method and test-class level as described in Section 3.2.3. Then we used the structured retrieval as described in Section 3.6. From the Figures 5 and 6, we see that structured retrieval works basically the same as unstructured retrieval at both test-method and test-class levels for LDiff. It shows a slightly better performance (median improvement of 0.02) for HDiff.

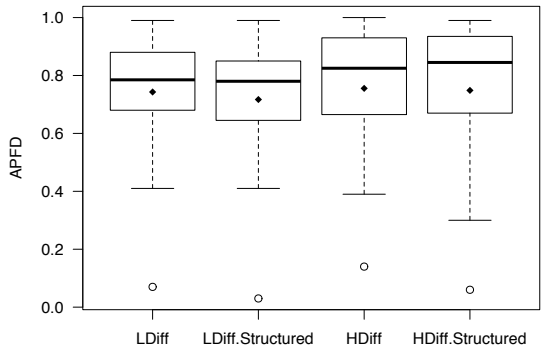

Figure 5: Impact of Structured Retrieval at test-method Level

Our results show that although structured retrieval has been found to improve the accuracy of bug localization results considerably, this is not the case for RTP.

4.4.4 Performance: REPiR Vs. JUPTA or Coverage-based RTP

To answer RQ4, first we ran all the eight techniques (four variants of JUPTA based on call graphs and four variants of coverage-based technique) described in Section 4.2 on each program-pair in our dataset. Figures 7 and 8 show the sum-

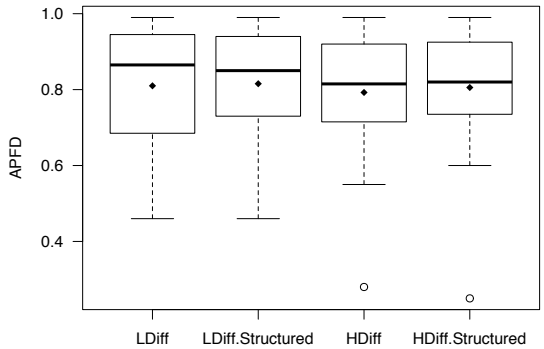

Figure 6: Impact of Structured Retrieval at test-class Level

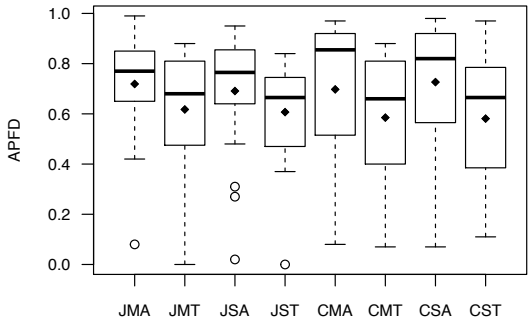

Figure 7: Performance of JUPTA and Coverage-based RTP at test-method Level

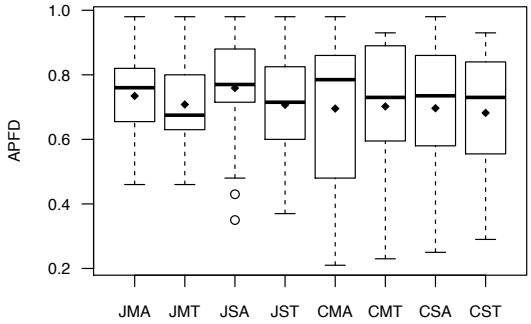

Figure 8: Performance of JUPTA and Coverage-based RTP at test-class Level

mary of APFD values for each strategy at method-level and class-level respectively. Results show that for both static and dynamic techniques, *additional* strategies are more effective than *total*, regardless of test-case granularities. This is consistent with prior studies [19, 53].

Now we compare all the mean APFDs of REPiR (from Figures 3 and 4) with that of JUPTA and coverage-based techniques (from Figures 7 and 8). From the results, we see that REPiR equipped with HDiff at method level and LDiff at class level overall outperformed both JUPTA and coverage-based approaches. At test-method level the mean APFD achieved by REPiR is 0.76, whereas the best variants of JUPTA (JMA) and coverage-based technique (CSA) achieve 0.72 and 0.73, respectively. At test-class level, REPiR equipped with LDiff achieves the mean APFD of 0.81, whereas the best variants of JUPTA (JSA) and coverage-best technique (CSA) achieve 0.76 and 0.7, respectively. Even for the compact representation of queries, REPiR performs better than any *total* strategies at test-method level (mean of 0.72 for HDiff.Distinct) and any *total* and *additional* strategies at test-class level (mean of 0.81 for LDiff.Distinct).

We further investigated how REPiR performs for each subject system. Each column in Tables 3 and 4 show the results of best configuration from each category (REPiR, JUPTA, and coverage-based). Results show that REPiR achieved the best APFD for five out of seven systems at test-method level and four subject systems at test-class level.

Finally, we perform statistical tests to investigate if the differences between various strategies are significant. Before applying paired significance test, we first apply the *Shapiro-Wilk Normality Test* to check the normality assumption and found that the accuracy achieved by different strategies are not normally distributed. Therefore, we perform *Wilcoxon*

Table 3: Comparison by Subjects test-method Level

Sub. Sys.	HDiff	JMA	CSA
Time and Money	0.50	0.47	0.19
Mime4J	0.68	0.68	0.59
Jaxen	0.67	0.67	0.94
XML-Security	0.80	0.42	0.69
XStream	0.84	0.68	0.79
Commons-Lang	0.95	0.79	0.86
joda	0.75	0.87	0.78

Table 4: Comparison by Subjects test-class Level

Sub. Sys.	LDiff	JSA	CSA
Time and Money	0.82	0.91	0.45
Mime4J	0.89	0.79	0.66
Jaxen	0.61	0.57	0.86
XML-Security	0.90	0.77	0.37
XStream	0.87	0.83	0.74
Commons-Lang	0.96	0.62	0.84
Joda-Time	0.63	0.66	0.69

Sign-Rank Test for each pair shown in Table 5. The input to the test is two vectors, where each vector consists of 20 APFD values for 20 version-pairs achieved by a given strategy. From the results we see that for any test-case granularity (either test-class or test-method), REPiR equipped with simply low-level program differences works significantly better than any *total* strategies (either static or dynamic). Also, as we have already noticed, high-level program differences works better at test-method level. Results show that at method-level, REPiR equipped with HDiff significantly outperformed any static *total* strategies even at a significance level at 0.01. On other hand, although REPiR performed better than the *additional* strategies as well in terms of mean, statistically the differences are not that significant.

Statistical test results for UT and RT: While answering RQ1 in Section 4.4.1, we did not perform any statistical test since we just wanted to get an idea how REPiR performs with respect to UT and RT. We have not also discussed different variants of REPiR by then. Thus, now we perform the same Wilcoxon test between REPiR results and UT and RT results at both test-case levels for both LDiff and HDiff. As shown in the last two rows of Table 5, REPiR significantly outperformed seven out of eight combinations of UT and RT.

Table 5: Wilcoxon Test Results (p-values)

	Method Level		Class Level	
Strategy	LDiff	HDiff	LDIff	HDiff
JMT	0.002**	0.002**	0.029*	0.048*
JMA	0.467	0.235	0.111	0.052
JST	0.003**	0.001**	0.001**	0.042*
JSA	0.391	0.145	0.171	0.201
CMT	0.01*	0.002**	0.048*	0.218
CMA	0.780	0.506	0.126	0.140
CST	0.011*	0.004**	0.015*	0.113
CSA	0.702	0.513	0.053	0.151
UT	0.151	0.03*	0.001**	0.004**
RT	0.015*	0.015*	0.0002**	0.0004**

* indicates significance at the 0.05 level ($p<0.05$)
**indicates significance at the 0.01 level ($p<0.01$)

```
public StringBuffer format(Calendar calendar,
        StringBuffer buf) {
-       if (mTimeZoneForced){
-               calendar.getTimeInMillis();
-               calendar = (Calendar) calendar.clone();
-               calendar.setTimeZOne(mTimeZone);
-       }
}
```

Figure 9: Faulty-edits in Commons-Lang 3.02

4.4.5 *Performance when REPiR is oblivious of Programming Language*

For this experimental setting, REPiR does not build any AST for source code or test classes while constructing document collections and queries. Documents are made at test-class level by simply removing mathematical operators and tokenizing any text that are in the test-classes. So the documents are expected to have noisy results because of language keywords. We used LDiff as query, which is also program language independent. Then we run REPiR for all version-pairs and calculate the APFD values. Results show that the mean APFD across all version-pairs is 0.8, while it was 0.81 when we used only identifiers and comments as document terms. The median difference is slightly more than that of mean (0.81 for language-oblivious configuration vs. 0.84 when we used only identifiers and comments).

4.5 Qualitative Analysis

Our hypothesis is that, in real-world software projects, developers tend to choose meaningful terms for identifier (e.g. classes, methods, variables) names and write comments in source code. It turns out that developers also use very similar terms for corresponding test cases. One of our main motivations of building an IR-based RTP is to exploit these common practices. In this section, we illustrate a concrete example to show the usefulness of this information.

When Commons-Lang evolved from version 3.02 to 3.03, test-method `FastDateFormatTest.testLang538` failed since the developer incorrectly removed a conditional block for updating time zone in method `FastDateFormat.format()` (shown in Figure 9, highlighted in red). If we extract the program differences from this change, LDiff produces the following terms: time, zone, forced, calendar, get, time etc., while HDiff produces `CM:FastDateFormat.format`. It should be noted there were also many other (non-faulty) changes in the query. Now let us take a look at the test-method that reveals this fault in Figure 10. Interestingly, we see many of the terms from faulty edits in the test-method. Furthermore, the source code class (`FastDateFormat`) and the corresponding test-class (`FastDateFormatTest`) have similar names. As a result, REPiR with HDiff ranked this method at 7th and LDiff ranked at 17th position among 1,698 test-methods. On the other hand, the best variants of JUPTA and coverage-based technique, JMA and CSA ranked it at 367th and 370th position respectively.

In spite of these extremely good results, there was an occasion, where REPiR performed unsatisfactorily. We investigated this case, and found that it was for the system, Time and Money, when it was evolving from 4.0 to 5.0. We found that the fault revealing test-method was `MoneyTest.testPrint` where there was apparently no information of use to IR. The only line in the test-method is `assertEquals("USD 15.00",d15.toString());`. However, our overall results show that this rarely happens in our subjects.

4.6 Time and Space Overhead

The running time of REPiR depends on three paramters: the size of vocabulary, the number of test cases, and the length of program difference. REPiR works most efficiently when we use compact representation of the query. It takes only a fraction of a second for each version-pair to prioritize its test cases. For example, REPiR took only 0.18 second to prioritize all the test-methods of Joda-Time 1.20, which is the largest system in our study. The preprocessing and indexing time is also approximately three seconds in total. When we used the full representation of query, REPiR took 20 seconds. On the other hand, the *additional* strategy based on statement level coverage information took 40 seconds, only for test prioritization (excluding instrumentation and coverage collection). The time complexity of REPiR and *total* strategy grows linearly when test suite size increases, while the *additional* strategy grows quadratically [34]. In addition, as discussed in Section 4.4.4, REPiR even with compact queries performs better than the *total* strategies and at least as effectively as the *additional* strategies. Thus, REPiR is a more cost-effective approach. Furthermore, note that coverage-based approach is not useful if the coverage information is not available from the old version because developers can simply run all the tests instead of spending time for recollecting the coverage. We performed all the experiments on a MacBook Pro with 2.8GHz and 4GB RAM.

The space overhead of REPiR is determined by the requirement of indexing test cases for IR. For most of the subject systems, index-size was around 1MB. For the largest system, Joda-Time 1.20, the index-size was 3MB. On the contrary, the data required by the traditional techniques for the same system was 11.6MB for method coverage matrix and 31MB for statement coverage matrix. The time and space overhead of JUPTA is very similar to coverage-based approaches since JUPTA tries to simulate code coverage.

4.7 Threats to Validity

This section discusses the validity and generalizability of our findings.

Construct Validity: We used two artifacts of a software repository: program source code and test cases, which are generally well understood. Our evaluation uses subject systems with real regression faults. Also we applied all the prioritization techniques on the same dataset, enabling fair comparison and reproducible findings. In order to evaluate the quality of prioritization, we chose Average Percentage Faults Detected (APFD), which has been extensively used in the field of regression test prioritization, and is straightforward to compute. APFD expresses the quality of prioritized test cases based on how early the faulty test cases are positioned in the prioritized suite. However, APFD does neither consider the execution time of individual test cases nor consider the severity of faults. Therefore, it may not accurately estimate how much we are gaining from the prioritized test suite in terms of cost and benefits.

Internal Validity: Since IR-based RTP works based on the term-similarity between the source code and test cases, the success of REPiR vastly depends on meaningful identifier names and comments, consistent with programming best practices.

External Validity: Our experimental results are based on 20 versions of programs from seven software projects, all of them are open source projects and written in Java.

```
public void testLang538() {
    final String dateTime = "2009-10-16T16:42:16.000Z";
    // more commonly constructed with: cal = new GregorianCalendar(2009, 9, 16, 8, 42, 16)
    // for the unit test to work in any time zone, constructing with GMT-8 rather than default locale time zone
    GregorianCalendar cal = new GregorianCalendar(TimeZone.getTimeZone("GMT-8"));
    cal.clear();
    cal.set(2009, 9, 16, 8, 42, 16);
    FastDateFormat format = FastDateFormat.getInstance("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", TimeZone.getTimeZone("GMT"));
    assertEquals("dateTime", dateTime, format.format(cal));}
```

Figure 10: Fault Revealing Test-Method for Commons-Lang 3.02

Although, they are popular projects and widely used in regression testing research, our findings may not be generalizable to other open source projects or industrial projects. Furthermore, all the subject systems in our experiment use JUnit test cases. Therefore, we cannot generalize our results for other types of tests. The risk of insufficient generalization could be mitigated by applying REPiR on more subject systems (both open source and industrial). This will be explored in our future work.

5. RELATED WORK

Reducing the time and cost of regression testing has been an active research area for near two decades. Researchers have already proposed various regression testing techniques, such as regression test selection [1, 39], prioritization [12, 34], and reduction [38]. Since our work is for regression test prioritization (RTP), this section is limited to the relevant work in this area. For related work regarding IR in software engineering, please refer to Section 2.2.

Wong et al. [50] introduced the notion of RTP to make regression testing more effective. They made use of program differences and test execution coverage from the previous version, and then sorted test cases in order of increasing cost per additional coverage. Rothermel et al. [40] empirically evaluated a number of test prioritization techniques, including both the *total* and *additional* test prioritization strategies using various coverage information. In that work, they also proposed the widely used APFD metric for test prioritization. Along the same line, Elbaum et al. investigated more code coverage information [13], and incorporated the cost and the severity of each test case for test prioritization [12]. Jones and Harrold [20] argued that there are important differences between statement-level coverage and modified condition/decision coverage (MC/DC) for regression testing, and proposed test reduction and prioritization using the MC/DC information. Jeffrey and Gupta [17] introduced the notion of relevant slices in RTP. Their approach assigns higher weight to a test case that has larger number of statements (branches) in its relevant slice of the output. However, a common limitation of these techniques is that they require coverage information for the old version, which can be costly to collect or not available in the repository.

Besides investigating different types of coverage information, researchers have also proposed various other strategies for RTP. Li et al. [25] used search-based algorithms, such as hill-climbing and genetic programming, for test prioritization. Jiang et al. [19] used the idea of adaptive random testing for test prioritization. Zhang et al. [53] recently proposed a spectrum of test prioritization strategies between the traditional *total* and *additional* strategies based on statistical models. However, according to the reported empirical results [19, 53], the traditional *additional* strategy remains one of the most effective test prioritization strategies.

There are also some approaches that do not require dynamic coverage information. Srikanth et al. [45] proposed a value-driven approach to system-level test case prioritization based on four factors: requirements volatility, customer priority, implementation complexity, and fault proneness of the requirements. Tonella et al. [48] used relative priority information from the user, in the form of pairwise test case comparisons, to iteratively refine the test case ordering. Yoo et al. [51] further used test clustering to reduce the manual efforts in pairwise test comparisons. Ma and Zhao [29] distinguished fault severity based on both users knowledge and program structure information, and prioritized tests to detect severe faults first. All these approaches require inputs from someone who are familiar with the program under test, which may be costly and not always available. To avoid manual efforts, Zhang et al. [55] proposed a static test prioritization approach, JUPTA, which extracts static call graph of a given test case to estimate its coverage. Later Mei et al. [34] extended the study and proposed more variants of JUPTA along the same way. Recently, Jiang and Chan [18] proposed a static test prioritization approach based on static test input information. However, all these approaches try to use static information to simulate code coverage, and thus may be imprecise. In contrast, REPiR is a fully automated (does not require user knowledge) and lightweight (does not require coverage collection or static analysis) test prioritization approach based on information retrieval, and has been shown to be more precise than many existing techniques.

6. CONCLUSION

To reduce the regression testing cost, researchers have developed various techniques for *prioritizing* tests such that the higher priority tests have a higher likelihood of finding bugs. However, existing techniques require either dynamic coverage information or static program analysis, and thus can be costly or imprecise. In this paper, we introduced a new approach, REPiR, to address the problem of regression test prioritization by reducing it to a standard IR problem. REPiR does not require any dynamic profiling or static program analysis. We rigorously evaluated REPiR using a dataset consisting of 20 version-pairs from seven projects with real regression faults, and compared it with 10 RTP strategies. The results show that REPiR is more efficient and performs significantly better than all *total* (dynamic or static) strategies while matching the accuracy of all *additional* strategies. We also show that REPiR can be made completely oblivious of the underlying programming language for test-class prioritization, almost without loosing any accuracy. We believe that this new approach to RTP represents a promising and largely unexplored new territory for investigation, providing an opportunity to gain new traction on this old and entrenched problem of RTP. Moreover, further gains might be achieved by investigating such IR techniques in conjunction with traditional static and dynamic program analysis, integrating the two disparate approaches, each exploiting complementary and independent forms of evidence regarding RTP.

7. REFERENCES

[1] T. Ball. On the limit of control flow analysis for regression test selection. In *ISSTA*, pages 134–142, 1998.

[2] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. In *ASE*, pages 151–154, 2010.

[3] B. Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[4] D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Increasing diversity: Natural language measures for software fault prediction. *JSS*, 82(11):1793–1803, 2009.

[5] D. Binkley and D. Lawrie. Applications of information retrieval to software development. *ESE (P. Laplante, ed.)*, 2010.

[6] D. Binkley and D. Lawrie. Applications of information retrieval to software maintenance and evolution. *ESE (P. Laplante, ed.)*, 2010.

[7] D. Binkley, D. Lawrie, and C. Uehlinger. Vocabulary normalization improves ir-based concept location. In *ICSM*, pages 588–591, 2012.

[8] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Metrics*. IEEE Computer Society, 2005.

[9] S. Deerwester, S. T. Dumais, G. W. Furn, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[10] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *ICSM*, pages 411–420, 2005.

[11] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *ISSRE*, pages 113–124, 2004.

[12] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*, pages 329–338, 2001.

[13] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *TSE*, 28(2):159–182, 2002.

[14] H. Fang, T. Tao, and C. Zhai. A formal study of information retrieval heuristics. In *SIGIR*, pages 49–56, 2004.

[15] W. Frakes. A case study of a reusable component collection in the information retrieval domain. *JSS*, 72(2), 2004.

[16] E. Hill, S. Rao, and A. Kak. On the use of stemming for concern location and bug localization in java. In *SCAM*, pages 184–193, 2012.

[17] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *COMPSAC*, pages 411–420, 2006.

[18] B. Jiang and W. Chan. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In *COMPSAC*, pages 190–199. IEEE Computer Society, 2013.

[19] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244. IEEE, 2009.

[20] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM*, pages 92–101, 2001.

[21] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

[22] D. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *ICPC*, pages 149–158, 2006.

[23] M. Lease, J. Allan, and W. B. Croft. Regression Rank: Learning to Meet the Opportunity of Descriptive Queries. In *ECIR*, pages 90–101, 2009.

[24] H. K. N. Leung and L. White. Insights into regression testing. In *ICSM*, pages 60–69, 1989.

[25] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *TSE*, 33(4):225–237, 2007.

[26] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE*, pages 234–243, 2007.

[27] A. D. Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *ICSM*, pages 299–309, 2006.

[28] S. Lukins, N. Kraft, and L. Etzkorn. Bug localization using latent dirichlet allocation. In *WCRE*, pages 155–164, 2010.

[29] Z. Ma and J. Zhao. Test case prioritization based on analysis of program structure. In *APSEC*, pages 471–478, 2008.

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

[31] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107–114, 2001.

[32] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–135, 2003.

[33] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. Using data fusion and web mining to support feature location in software. In *ICPC*, pages 14–23, 2010.

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

[35] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual ACM SIGIR conference*, pages 275–281, 1998.

[36] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.

[37] S. E. Robertson, S. Walker, and M. Beaulieu. Experimentation as a way of life: Okapi at trec. *IPM*, 36(1):95–108, 2000.

[38] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*, pages 34–43, 1998.

[39] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[40] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *ICSM*, pages 179–188, 1999.

[41] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[42] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *IPM*, 24(5):513–523, 1988.

[43] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *FSE*, pages 297–298, 2009.

[44] A. Singhal. Modern information retrieval: A brief overview. *DEB*, 24(4):35–43, 2001.

[45] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *ISESE*, pages 64–73, 2005.

[46] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. In *ICIA*, pages 2–6, 2005.

[47] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, pages 253–262, 2011.

[48] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *ICSM*, pages 123–133, 2006.

[49] X. Wei and W. B. Croft. Lda-based document models for ad-hoc retrieval. In *ICRDIRl*, pages 178–185, Seattle, WA, 2006.

[50] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE*, pages 230–238, 1997.

[51] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ISSTA*, pages 201–212, 2009.

[52] C. Zhai. Notes on the lemur tfidf model (unpublished work). Technical report, Carnegie Mellon University, 2001.

[53] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*, pages 192–201, 2013.

[54] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In *FSE*, pages 40:1–40:4, 2012.

[55] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing JUnit test cases in absence of coverage information. In *ICSM*, pages 19–28, 2009.

[56] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.