2014 21st Asia-Pacific Software Engineering Conference

Test Case Prioritization Based on Information Retrieval Concepts

Jung-Hyun Kwon, In-Young Ko
Dept. of Comp. Science
KAIST, Daejeon, South Korea
{arunson, iko}@kaist.ac.kr

Gregg Rothermel
Dept. of Comp. Science
Univ. of Nebraska-Lincoln, NE
grother@cse.unl.edu

Matt Staats
SnT Centre
Univ. of Luxembourg, Luxembourg
staatsm@gmail.com

***Abstract*—In regression testing, running all a system's test cases can require a great deal of time and resources. Test case prioritization (TCP) attempts to schedule test cases to achieve goals such as higher coverage or faster fault detection. While code coverage-based approaches are typical in TCP, recent work has explored the use of additional information to improve effectiveness. In this work, we explore the use of Information Retrieval (IR) techniques to improve the effectiveness of TCP, particularly for testing infrequently tested code. Our approach considers the frequency at which elements have been tested, in additional to traditional coverage information, balancing these factors using linear regression modeling. Our empirical study demonstrates that our approach is generally more effective than both random and traditional code coverage-based approaches, with improvements in rate of fault detection of up to 4.7%.**

I. Introduction

In regression testing, existing test cases are executed to check whether the software still works correctly after modifications. Running all test cases is the simplest way to perform regression testing; however, this may require a great deal of time and computing resources. Existing regression testing methods such as *test case selection*, *test case prioritization* and *test suite reduction* [1] attempt to make regression testing more cost-effective.

The focus of this work is test case prioritization (TCP), which attempts to schedule test cases in order to achieve goals such as faster code coverage or higher rates of fault detection. TCP is typically required when there are many test cases but testing time is limited. Modern software development is characterized by frequent software updates, underlining the importance of TCP.

Previous TCP techniques have focused largely on *code coverage information* such as line and branch coverage, since a test case having greater code coverage may naturally be expected to find more defects in the system under test. However, in some cases, code coverage is not a good predictor of the fault detection ability of a testing process [2], [3]. In such cases, code coverage-based techniques for prioritization may not be especially effective.

Recent work has explored the use of other information along with the code coverage for prioritizing test cases. For example, there is research on utilizing software requirements or human assistance [4], [5]. However, there has been little research focusing on *code rarely executed* by existing test cases. Leon et al. [6] point out the importance of considering the frequency at which code is tested. Examples of less tested code include exception handling code and code handling unusual conditions. Putting weight on test cases executing such code regions may increase the rate at which faults are detected.

In this paper, we propose an alternative prioritization technique that adapts *Term Frequency (TF)* and *Inverted Document Frequency (IDF)* to overcome the limitations of code coverage-based prioritization techniques. TF and IDF are basic concepts in the Information Retrieval (IR) field. We believe that such an approach may find defects more quickly by considering not only code coverage information but also how many times a coverage element has been executed by a test case (TF) and source code elements tested by few test cases (IDF). Our approach adopts a linear regression model to automatically determine how to weigh the value of these two sources of information during prioritization. As in most TCP approaches, each test case is given a value and the test cases are scheduled according to the values.

To evaluate the effectiveness of our approach, we conducted an experiment with four Java programs, of varying size and with varying sets of test cases. As baseline approaches for comparison, a random ordering, and common code coverage-based techniques are used. The effectiveness of each technique is measured in terms of *Average Percentage of Fault Detection (APFD)*, which indicates fault detection rate. Our results demonstrate that our approach is generally more effective than random and code coverage-based approaches – the approach increases the rate of fault detection by up to 4.7%, with 16 of 19 improvements being statistically significant. This study suggests that advanced IR techniques or regression models can be applied to TCP.

The remainder of this paper is organized as follows. Section 2 provides background information on TCP and IR. Section 3 presents our TF-IDF-based approach. Section 4 presents our experiment design. Section 5 presents and analyzes our experiment results. Section 6 overviews related work and Section 7 presents conclusions and future work.

II. Background

A. Test Case Prioritization

TCP attempts to schedule test cases to achieve certain goals [7]. Prioritized test cases may help a tester achieve a requested coverage criterion faster in the testing process. Prioritized test cases may also help a tester find regression faults faster than unordered ones. Crucially, this can allow the tester to begin debugging faster and consequently the technique can provide more confidence in terms of the reliability of the software. Rothermel et al. [7] define TCP as follows:

Definition: Given T, a test suite, PT, the set of permutations of T, and f, a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'')\ (T'' \in PT)\ (T'' \neq T'$ $[f(T') \geq f(T'')]$.

To increase the value of function f, various TCP techniques have been proposed. Traditionally, code coverage-based techniques have be used, under the notion that a test case covering a lot of code can detect faults in a program with a higher probability than other test cases covering less code. Typical code coverage-based techniques involve line (statement) coverage, branch coverage, and method coverage.

In practice, code coverage-based TCP techniques come in two flavors: those based on *total-coverage* and those based on *additional-coverage* [7]. In total-coverage techniques, test cases are compared to one another with respect to their total number of covered code elements. Additional-coverage techniques add a feedback mechanism; when deciding on a next test case these techniques consider coverage of unscheduled test cases with regard to code elements not yet covered.

B. *Information Retrieval*

Information Retrieval (IR) is an activity conducted to find documents satisfying a user's information needs given an unstructured document collection [8]. The information needs are represented as queries. Term Frequency (TF) and Inverted Document Frequency (IDF) are statistics indicating how important each word in a document or query is to the document or query. TF represents how many times a word appears in a document or query; as the TF score of a word increases, the word appears more often in the document or query. The second concept, IDF, is an inverse of Document Frequency (DF). The DF of a word indicates the number of documents in the collection containing the word. Therefore, if a word has a large IDF score, the word does not often occur in other documents and is relatively unique. By measuring TF and IDF scores for each word, a similarity between a document and a query is measured. Words having high TF or IDF scores in a document or query can be considered as representative words of the document or query.

The similarity between a document and a query can be measured by the sum of TF $\times$ IDF scores of common words between the document and query. The formal equation for the similarity score is as follows [8]:

$$Similarity\,Score(q,d) = \sum_{t \in q \cap d} tf\text{-}idf_{t,d}$$

In this equation, q refers to a query, t is a term and d is a document. When there are many common words between a document and query, or when the TF and IDF values of an individual word are high, the score increases.

III. Proposed Approach

A. *Motivation*

Code coverage-based TCP techniques can be useful when there is a positive correlation between the code coverage of

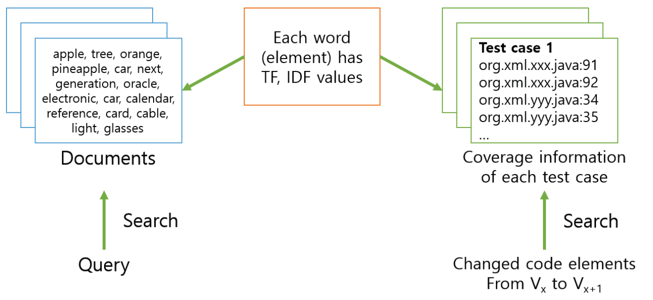

apple, tree, orange, pineapple, car, next, generation, oracle, electronic, car, calendar, reference, card, cable, light, glasses
Documents
Each word (element) has TF, IDF values
Test case 1
org.xml.xxx.java:91
org.xml.xxx.java:92
org.xml.yyy.java:34
org.xml.yyy.java:35
...
Coverage information of each test case
Search
Query
Search
Changed code elements From V_x to V_x+1

Fig. 1: Relationship Between TF-IDF and TCP

test cases and faults in the program. However, code coverage techniques are naturally not useful in the presence of weak correlation [2], [3]. Recent research proposes hybrid approaches combining code coverage with other aspects related to programs such as time and software requirements (e.g., [4], [9]). However, there has been little research focusing on less tested code—code that is tested, but with uneven frequencies. Such code is often responsible for key error handling functionality and can be crucial to application correctness. To overcome the limitation of coverage-based techniques and focus on less tested code, our proposed approach uses TF-IDF concepts from IR and linear regression in machine learning.

Fig. 1 shows how the concept of IR can be utilized in TCP. As shown, a document consists of words. For the purposes of TCP, however, we view a "document" as a test case, and the "words" as the elements covered (e.g., branches, lines, methods) by each test case. Likewise, while a query in IR consists of words, a query in this context is coverage elements in the updated files. Each covered element thus has TF and IDF scores, with the scores of an element covered by a test case indicating how important the element is to the test case. Searching relevant documents for a query in IR is carried out based on the TF and IDF scores of a word. In a similar way, searching relevant test cases for a query in TCP is performed based on TF and IDF scores of a covered element.

The TF score in TCP is related to the frequency at which a code element is exercised by a test case. As a test case executes a certain code element more times, the test case achieves a larger TF score on that element. The TF concept may be useful for finding faults caused by running certain code many times. For example, exercising a branch several times may increase the chance of visiting other program states. In this sense, test cases covering high TF code elements may have more capacity to detect faults in the program than other test cases, even in the presence of similar traditional coverage (i.e., similar branch coverage).

The DF score of a word in IR is the number of documents containing the word. In the context of TCP, the DF score of a covered code element is related to the number of test cases exercising the element. As the number of test cases covering an element increases, the DF score of the element increases. IDF is the inverse of DF. Thus, IDF scores can be used to find unique code elements. Test cases that cover many high IDF elements may have high capacity to find defects in the program because they cover many unique elements.

The similarity score between a test case (document) and changed elements (query) can be calculated by the sum of

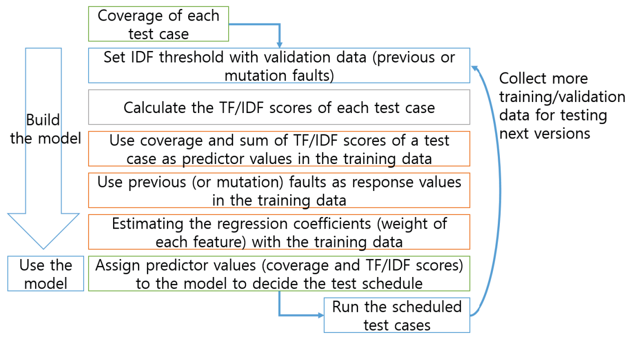

Coverage of each test case
Set IDF threshold with validation data (previous or mutation faults)
Calculate the TF/IDF scores of each test case
Use coverage and sum of TF/IDF scores of a test case as predictor values in the training data
Use previous (or mutation) faults as response values in the training data
Estimating the regression coefficients (weight of each feature) with the training data
Assign predictor values (coverage and TF/IDF scores) to the model to decide the test schedule
Run the scheduled test cases
Build the model
Use the model
Collect more training/validation data for testing next versions

Fig. 2: Procedure to Construct the IRCOV Model

the TF-IDF scores of the common elements between covered elements of the test case and elements in the query. The similarity score indicates how much the test case is related to the modifications.

B. Approach Overview

The goal of our approach is to improve the rate of fault detection of coverage-based TCP techniques. Towards this, we combine traditional coverage scores with similarity scores produced using IR. We use a linear regression model with two features to automatically set weights on the coverage and similarity scores of test cases. This regression model adjusts the weights of each feature such that a feature having little correlation with faults in a program has little weight.

Fig. 2 shows an overall process for building a *linear regression model* with IR and coverage information (IRCOV). The process consists of validation, training and test phases. In the validation and training phases, faults that were detected in previous versions can be used as validation and training data. In the validation phase, the IDF threshold is determined. Next, in the training phase, weight values are calculated in a manner that minimizes the learning error (difference between estimated and real values). In the test stage, the scheduled test cases are run and the fault detection effectiveness of the test cases is measured. We describe each step in detail in the following subsections.

C. Setting an IDF Threshold with Validation Data

An IDF threshold is the maximum number of test cases considered when assigning an IDF score to a code element. This property is required because a test case having high coverage can cover many *small IDF* elements. Since, the similarity score of a test case is the sum of the (TF $\times$ IDF) scores of code elements, the test case may have a large similarity score. This is not relevant to our objectives, as our focus is on detecting faults in less tested elements faster.

To avoid the foregoing problem, we set an IDF threshold. The IDF threshold is decided by the validation data. The validation data consist of faults that existed in previous versions and information about which test cases detected the faults. For a ratio from 0.1 to 1.0, the threshold is set by the number of test cases $\times$ the ratio. With each threshold, the regression model is trained with the validation data. Only code elements whose DF is not above the threshold can obtain an IDF value. After the models with each ratio are built, their learning errors are compared. The ratio that leads to the minimum learning error is selected for the IDF threshold. When there are multiple candidate ratios, our approach selects the smallest ratio. The model is trained with validation data in the same manner as with training data. Details on the training process are provided in the following subsections.

D. Calculating TF-IDF Scores of Each Test Case

In the study reported on in this paper, we calculated TF-IDF scores based on TF-IDF schemes from the SMART Information Retrieval System where various TF and IDF types are defined [10]. This study focuses on less tested code, which is related to IDF. Thus, for TF scores, Boolean type is selected from the schemes, which reduces the impact of TF. For IDF scores, logarithm type is selected from the schemes (a widely used type in the IR field). Thus, the TF score of a code element is 1 if the element is covered; otherwise, it is 0. In addition, we define the IDF score of a code element as $1 + log(\frac{\#\, of\, test\, cases}{\#\, of\, test\, cases\, covering\, the\, element})$.

The similarity score of a test case is determined by the sum of the TF-IDF scores of common covered elements e between the test case t and elements in the changed files q, as follows:

$$Similarity\, Score(t, q) = \sum_{e \in t \cap q} tf\text{-}idf_{e,t}$$

E. Constructing the IRCOV Model

The linear regression equation we use is defined as follows:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

In the equation, there are two predictors (features). x_1 denotes the size of coverage data for each test case and x_2 refers to the similarity score of each test case. We define y as the *fault detecting capability* of the test case. The capability is proportional to the number of previous faults detected by the test case. In addition, each fault is scaled to allow our regression model to detect rare faults as well as many faults. Rather than assigning each fault a value of 1, we assign it the value of 1 divided by the number of test cases detecting the fault. We apply a log function to the number of test cases to prevent the fault capability from decreasing too much. The y value of a test case is thus defined as follows:

$$y = \sum_{i=1}^{n} \frac{f_i}{log(t_i) + 1}$$

Here, n is the total number of previous faults. The value of f_i is 1 if the test case detected fault i, otherwise, it is 0. t_i refers to the number of test cases detecting fault i.

In the linear regression equation presented above, theta denotes a weight for each feature. θ_0 is for the intersect of the linear equation. To find appropriate theta values, a set of training data is needed. The training data consists of predictor values x and response values y, and can be obtained from previous fault information. Once x values and y values have been calculated, theta values can be calculated by the following *normal equation* [11]:

$$theta = (X^T X)^{-1} X^T \vec{y}$$

F. Prioritizing Test Cases with the IRCOV Model

Finally, the theta and feature values are used to prioritize the test cases. The linear model produces an estimated number of faults that each test case can detect. Then, the test cases are ordered in terms of descending estimated values.

Note that in the IRCOV model, *total-coverage* and *additional-coverage* techniques have different x_1 and x_2 values in the regression model. In the total-coverage technique, coverage and similarity values of test cases remain constant. In the additional-coverage technique, however, the feedback changes the remaining uncovered code elements; thus, coverage and similarity values change as test cases are scheduled. When calculating the similarity value of a test case, the sum of TF $\times$ IDF is calculated based on the remaining uncovered code elements and the query.

IV. Evaluation

A. Research Questions

To understand whether TF-IDF scores can increase the fault detection effectiveness of the coverage-based techniques, we explore the following research questions:

RQ1: Is IRCOV more effective than random?
RQ2: Is IRCOV more effective than the coverage-based techniques?

RQ1 is explored to determine whether our approach is at least better than random ordering, and more importantly to determine whether scheduling test cases has any value for the object programs studied. If a random approach performs similar to other techniques, including traditional coverage-based techniques, this signals that TCP is a poor fit for these objects. RQ2 is explored to determine whether our approach is more effective than the most commonly used TCP techniques.

B. Objects of Study

For this study, we selected four small and middle-size Java open source programs. Xml-Security is from the software-artifact Infrastructure Repository (SIR) [12] and all other objects were used previously in a test oracle generation study [13]. All the objects provided JUnit-based test suites.

TABLE I: Object Information

	Versions	LOC	Test Classes	Test Methods	Mutants
CCL	1.0, 1.1, 1.2	875	21	107	1012
CCN	3.0, 3.1, 3.2	18200	200	3584	7589
JOD	2.0, 2.1, 2.2, 2.3	14315	121	3586	12585
XSE	0, 1, 2, 3	11081	15	94	7929

Table I describes relevant information for each object considered in the experiment. The first column provides a mnemonic for each object. CCL (Commons-CLI) is a library providing an API parsing command line arguments. CCN (Commons-Collections) provides Java data structures. JOD (Joda-Time) is a library providing Java data and time API. XSE (Xml-Security) provides Java XML digital signature, XML encryption, etc. The second column indicates the evaluated versions. The third column lists the number of lines of Java source code in each object, as measured by the *JaCoCo* coverage tool.[1] The fourth and fifth columns include the number of test cases at the class-level and method-level, respectively. The sixth column describes the number of generated mutants for each object. The mutants are obtained from the MAJOR mutation tool [14].

C. Independent Variable and Treatments

The independent variable in our study is the prioritization technique used. To investigate the effectiveness of our approach, we chose seven baseline approaches. First, we created a random method, that schedules test cases using random selection. Next, as noted, we selected representative total- and additional-coverage approaches, using line, branch, and method coverage. Thus, three total coverage and three additional coverage techniques were used.

We built IRCOV models based on the six baseline coverage approaches. Thus, six IRCOV approaches were generated. We compare IRCOV approaches to random for RQ1 and to the baseline coverage approaches for RQ2. Table II lists the abbreviations and full names of the treatment approaches.

TABLE II: Treatments

Abbreviation	Full name
random	Random ordering
b	Total branch coverage
l	Total line coverage
m	Total method coverage
cBIR	IRCOV using total branch coverage
cLIR	IRCOV using total line coverage
cMIR	IRCOV using total method coverage
aB	Additional branch coverage
aL	Additional line coverage
aM	Additional method coverage
cBIR	IRCOV using additional branch coverage
cLIR	IRCOV using additional line coverage
cMIR	IRCOV using additional method coverage

D. Dependent Variable

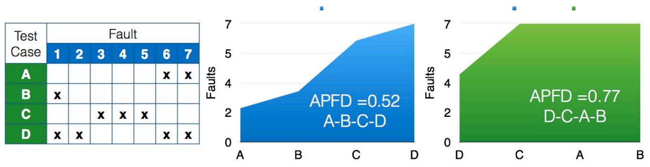
To measure the effectiveness of each prioritization approach we used the APFD metric. This metric indicates how quickly faults are detected during a run of a test suite. APFD values range from zero to one. As APFD nears 1, this indicates that the prioritization technique can find faults using fewer test cases. APFD is calculated as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + ...TF_N}{nm} + \frac{1}{2n}$$

In this equation, m is the number of faults in a program and n is the number of test cases. TF_i refers to an ordering index of a test case that detects a fault i in the scheduled test suite.

Fig. 3 provides an example of APFD. There are four test cases, A, B, C, and D, and each test case detects several of seven faults. When no prioritization technique is applied (A-B-C-D), TF_i values are 2,4,3,3,3,1 and 1, which yields an APFD value of 0.52. However, for test cases in order D-C-A-B, TF_i values become 1,1,2,2,2,1 and 1, yielding a higher (better) APFD value of 0.77.

[1]http://www.eclemma.org/jacoco/



Test Case	Fault 1	2	3	4	5	6	7
A						x	x
B	x						
C			x	x	x		
D	x	x				x	x

Fig. 3: APFD illustration

E. Controlled Factors

1) Test Suites: We use test suites provided with each object program. Test cases that can be processed by the coverage tool and mutation tool are considered in TCP. In this study, class-level test cases are used. There are pros and cons for each possible level of test-case granularity in the context of TCP [15]. In JUnit testing, running individual method-level test cases may achieve faster fault detection, but requires longer testing time because of setup and tear down code in a class-level test case. This code can include database or network connection and data initialization code. Method-level test cases run such code in every test case, which can be expensive. Class-level test cases do not have this problem, and represent a typical testing approach; thus we use class-level test cases in this work.

2) Version Change Information: TCP techniques are used in regression testing, so each object must have multiple previous versions and their test suites. We obtained multiple versions of each object from SIR and the objects' source code repositories. The proposed approach requires changed code information, which we collected as follows. First, we used the *Ubuntu diff* [2] utility to identify changed Java files between two versions. This *diff* tool can ignore blank lines and spaces. Next, we used Eclipse JDT (Java Development Tools)[3] to parse Java code and remove JavaDoc and comments. Then, we used the *diff* tool again to make sure that differences between files were caused by code changes.

3) Faults: The fault information provided with each version of our object programs is not sufficient to allow us to evaluate our approach, so we simulate faults by using mutation faults. From the mutants generated by the mutation tool, we collected killed mutants. Killed mutants are mutants that result in failures on one or more test cases. We use these mutants as training, validation and test data for our approach.

Mutation faults can be of various types. For example, the arithmetic operator "+" can be mutated into "-" or "*". MAJOR supports eight mutation operation types. The mutation operators are: Arithmetic operator replacement, Logical Operator Replacement, Conditional Operator Replacement, Relational Operator Replacement, Shift Operator Replacement, Operator Replacement Unary, Statement Deletion Operator and Literal Value Replacement [14].

Given a set of killed mutation faults, we randomly shuffled the mutants and partitioned them into validation, training and test data. Since obtaining mutation faults is an expensive process, using many mutation faults to build the regression model can be unrealistic. Thus, we used 10% of them for validation, 10% for training, and 80% as test data. However, using all the test data in one faulty version would be unrealistic. Therefore, as in [16], we generated 50 fault subsets from the test data. Each subset contains 5-15 mutation faults and serves as a faulty version.

Note that the way we partition mutation faults can be a cause of bias. In other words, APFD values resulting from test cases prioritized by our approach might be closely related to the chosen mutation faults. To reduce the potential for bias, we applied 10-fold validation to the mutation faults. 10-fold validation is a widely used technique in the machine learning field for achieving more reliable results. In each round, the mutation faults are divided into 10 subsets and each subset is assigned as training, validation or test data. As each round proceeds, this technique makes sure that every subset is used for training, validation or test data at least once.

F. Experiment Steps

The experiment was conducted in the following steps:

1) Partition the mutants into training, test and validation sets (10%, 80% and 10%, respectively).
2) Use the validation mutants to determine the IDF threshold.
3) Use the training mutants to determine the weights of coverage and similarity scores.
4) Generate 50 faulty versions of each object program by using the test mutants (one faulty version contains 5-15 faults).
5) Perform prioritization with each technique.
6) Measure the APFDs of each prioritized test suite against the faulty versions.
7) Determine the statistical significance of the APFD improvements using the Wilcoxon signed rank test.[4]
8) Repeat steps 2-7, 10 times (10-fold validation).

G. Threats to Validity

External validity. The objects used in the evaluation are small and medium sized Java programs. Thus, the objects, and their corresponding test cases, do not represent all possible applications. However, the number of test cases for each object varies and the object size also varies, providing some variation and improving generality.

Construct validity. APFD is used to measure the effectiveness of various prioritization techniques. There are other metrics such as considering the cost of each test case [17]. The results using different metrics may vary. However, APFD is a widely used metric and at least indicates how fast the technique finds program faults.

Mutation faults may not represent real faults. However, Andrews et al. [18] and Do et al. [19] present empirical results that indicate that real faults and hand-seeded faults can be replaced with killed mutants in testing experiments.

Internal validity. Linear regression is a basic and widely used predictive model in machine learning. However, the model assumes a linear relationship between the predictors and response. If non-linearity exists, the model's accuracy can

[2]http://manpages.ubuntu.com/manpages/hardy/man1/diff.1.html
[3]http://www.eclipse.org/jdt/

[4]http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

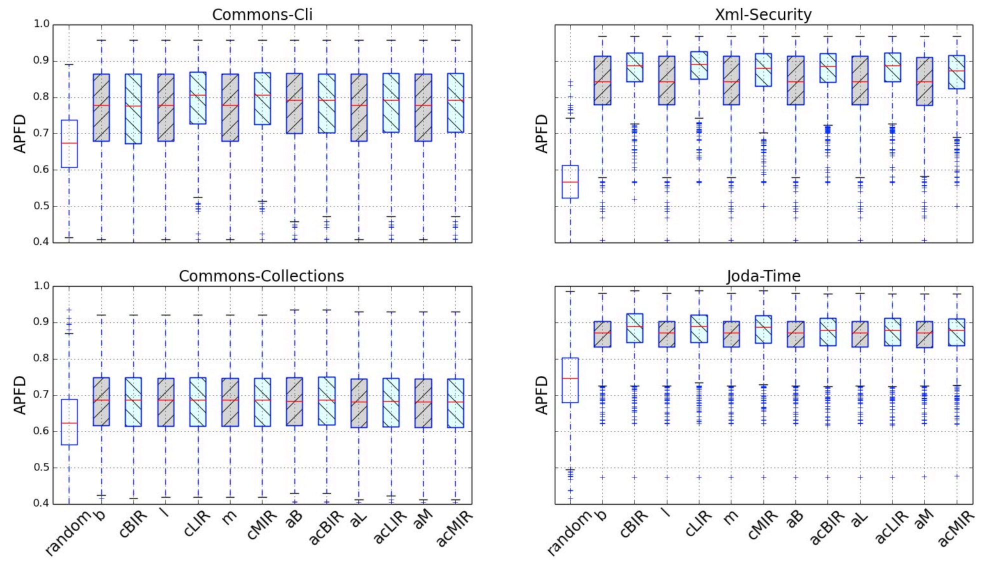


Fig. 4: APFD Boxplots for Each Technique and Object
Light Gray (Slash) Boxes: Coverage Based Techniques, Light Cyan (Backslash) Boxes: IRCOV Techniques

be reduced. To support various programs and faults, non-linear transformation such as $logX$, $\sqrt{X}$ and X^2 can be used.

Most of the techniques we used were developed in-house, and errors in our implementations could have led to erroneous results. However, the development of the traditional prioritization techniques was not complicated and was done in conjunction with individuals familiar with their construction.

V. Results and Analysis

In this section, we examine the results of our experiment in the context of our two research questions: *RQ1*, " Is IRCOV more effective than random?", and *RQ2*, "Is IRCOV more effective than coverage-based techniques?".

Fig. 4 presents boxplots showing APFD results for each technique and object program. The Y-axes denote APFD values, and each box represents the data derived from 50 subsets of test data across the 10 fold validation. The X-axes refer to techniques using mnemonics defined in Table II.

Table III presents median APFD values for each technique and object program, and results of statistical tests applied to the data. Table (a) presents results for RQ1, comparing median APFD values for each of our techniques with those of the random ordering. Table (b) presents results for RQ2, comparing each of our techniques with the corresponding coverage-based baseline technique. To determine whether the differences are statistically significant, we used Wilcoxon signed rank tests, setting the significance threshold to 0.001. As shown, nearly all differences are statistically significant. (The notation "-" means that the APFD values of two techniques are identical.)

We begin by examining each object program in turn. The plots in the upper left corner of Fig. 4 present results for CCI. In this case, all IRCOV approaches are more effective than random (imp: 10.2%, 13.2%, 13.1%, 11.7%, 11.7%, 11.7%). In addition, all of the improvements in effectiveness are statistically significant. Four of the six IRCOV techniques (cLIR, cMIR, acLIR, acMIR) are more effective than the traditional coverage-based techniques (imp: 2.78%, 2.72%, 1.33%, 1.33%), all with statistical significance.

The plots in the upper right corner of Fig. 4 present results for XSE. Here, all IRCOV techniques are better than random by a wide margin (imp: 32.0%, 32.4%, 31.4%, 31.9%, 32.0%, 30.7%), with statistical significance. Also, all IRCOV techniques outperform the corresponding coverage-based techniques by statistically significant margins (imp: 4.3%, 4.7%, 3.7%, 4.2%, 4.3%, 3.0%). More notably, IRCOV techniques exhibit less variance than the baseline techniques, indicating that the techyniques are more consistent than the coverage-based techniques.

The plots in the lower left corner of Fig. 4 present results for CCN. Here, all IRCOV techniques are more effective than random (imp: 6.4%, 6.4%, 6.3%, 6.3%, 5.9%, 5.8%), with statistical significance. Also three of the IRCOV techniques are slightly more effective than their coverage-based counterparts (imp: 0.1%, 0.3%, 0.1%); however, no differences are statistically significant.

Finally, the plots in the lower right corner of Fig. 4 present results for JOD. Here, similar to results on XSE, all IRCOV techniques are better than random (imp: 14.2%, 14.2%, 14.1%, 13.1%, 13.2%, 13.2%). Also, the IRCOV techniques are more effective than their corresponding coverage-based techniques (imp: 1.7%, 1.7%, 1.5%, 0.6%, 0.7%, 0.7%). All differences are statistically significant.

Note that the results of additional coverage techniques were similar to results for total coverage techniques. A possible reason for this may be related to code elements covered in the prioritization process. When the additional techniques schedule test cases, the remaining numbers of test cases and

		cBIR	cLIR	cMIR	acBIR	acLIR	acMIR
CCI	Imp(%)	10.24%	13.15%	13.10%	11.71%	11.71%	11.71%
	P-val	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
XSE	Imp(%)	32.00%	32.38%	31.43%	31.90%	32.00%	30.67%
	P-val	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
CCN	Imp(%)	6.36%	6.37%	6.29%	6.27%	5.93%	5.82%
	P-val	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
JOD	Imp(%)	14.17%	14.17%	14.06%	13.14%	13.16%	13.17%
	P-val	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001

(a) Comparison of Random & IRCOV Techniques for RQ1

		cBIR	cLIR	cMIR	acBIR	acLIR	acMIR
CCI	Imp(%)	-0.14%	2.78%	2.72%	0.00%	1.33%	1.33%
	P-val	$2.29E$-03	< 0.001	< 0.001	$1.70E$-01	< 0.001	< 0.001
XSE	Imp(%)	4.29%	4.67%	3.72%	4.20%	4.29%	2.97%
	P-val	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
CCN	Imp(%)	0.00%	0.08%	0.00%	0.25%	0.12%	0.00%
	P-val	$7.87E$-01	$1.02E$-01	-	$1.34E$-02	$1.67E$-03	-
JOD	Imp(%)	1.65%	1.65%	1.54%	0.62%	0.65%	0.65%
	P-val	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001

(b) Comparison of Traditional Coverage-Based & IRCOV Techniques for RQ2

TABLE III: APFD Improvements and P-values

the remaining code are updated. However, when test cases that are not scheduled do not cover any of the remaining code, we erase history information relative to the covered code and run the technique again. When test cases are scheduled the history is often reset. This may cause the additional techniques to perform similar to the total techniques.

Another notable result is that the weights of each feature have varying values across versions and faults. For CCI, the coverage weight, θ_1, ranges from 0.94 to 4.27, while one of the similarity scores, θ_2, ranges from -2.06 to 2.98. In the case of XSE, θ_1 ranges from -28.42 to 4.71, whereas θ_2 ranges from 9.36 to 34.97. For CCN, θ_1 ranges from -0.27 to 2.50, whereas θ_2 ranges from 0.57 to 2.66. Last, for JOD, θ_1 ranges from 3.53 to 11.66, while θ_2 ranges from -3.17 to 2.35. Most of the weight values for the similarity score are neither zero nor near zero. Therefore, we can clearly say that the IR information affected fault detection capability in the IRCOV model.

In conclusion, the IRCOV approaches generally performed more effectively than random and coverage-based techniques. There are 48 comparisons between the proposed techniques and baseline approaches for each object. In 43 of these 48 cases IRCOV techniques were better than baseline techniques, with 40 of the 43 improvements being statistically significant.

Thus, in the cases studied, IRCOV clearly improved over traditional techniques, offering consistent—albeit sometimes slight—improvements in fault detection rate, while being always at least as good as the traditional techniques. Note that this property of being always at least as good as coverage-based techniques is by design—the automated weighting provided by the linear regression modeling is crucial here, allowing us to rely more on IR techniques when the correlation between coverage and fault finding is low, and less otherwise. We therefore expect to never do worse when using IRCOV, but as shown, when opportunities exist to outperform traditional coverage we expect IRCOV to do so.

VI. Related Work

Many TCP techniques combining code coverage data with additional information have been proposed. Examples of additional information are software requirements, execution profiles, and human knowledge [4]–[6], [9], [20]. Also, many techniques use concepts from machine learning such as clustering, case-based ranking and Bayesian networks. In addition, IR techniques have been used in TCP [21], [22]. We elaborate on this prior work next.

Arafeen et al. [4] propose a prioritization technique using software requirements. They cluster the requirements by a similarity measure, and select test cases from each cluster following specific methods. Leon et al. [6] use cluster filtering with the additional coverage technique. Cluster filtering is based on test execution profiles measured in terms of covered code, data flow, and so forth. They analyze the distribution of the profiles and cluster them by a similarity measure. In relation to our approach, clustering can be used to identify test cases covering unique code. However, deciding on the number of clusters and size of each cluster can be an issue.

Jeffrey et al. [20] suggest the use of relevant slices, the subset of program lines that influence or potentially influence program output, combining the relevant slice when a program has changed with branch coverage. The authors give a higher weight to the relevant slice than to coverage of a test case when prioritizing test cases. Our approach differs, in that we suggest another weighting mechanism by using a linear regression model so that weights can be changed according to characteristics of programs and faults.

Mirarab et al. [9] explore the use of Bayesian networks for prioritization. They build a Bayesian network model based on information such as modified source code, fault-proneness and test coverage. Tonella et al. [5] increase the effectiveness of test case prioritization by adopting user knowledge. When there are cases where data is not enough to rank two test cases, human knowledge is used to give one of the test cases a higher priority. They use Case-Based Ranking, a machine learning algorithm, to combine multiple models. Similar to our approach, they use concepts in machine learning. However, they do not consider less tested code directly in their models.

Two recent approaches use IR techniques in TCP. Nguyen et al. [21] suggests the use of TCP for web service compositions. The authors propose a prioritization technique to give a higher priority to test cases that are related to web service changes. They measure the similarity between an execution trace of source code and the update description of a service to find faults caused by the service update early. Thomas et al. [22] suggest a static TCP using LDA, which is an advanced IR-based technique. They calculate the similarity between pairs of test cases so that test cases that target different functionalities are assigned higher priority. Both approaches use IR techniques to measure similarities, as our approach does. However, their approaches analyze literals that are extracted from code as similarity elements, whereas our approach utilizes lines of code. In addition, the domain of the former approach is restricted to applications that use web services. The latter approach is designed for black-box testing, so the work does not include any comparison to execution-based TCP techniques.

VII. Conclusion

In this paper, we address limitations of coverage-based TCP techniques; specifically, coverage does not always relate to fault detection, and the frequency with which code is tested has gone largely unconsidered in the prioritization process. To overcome these limitations, we have presented an approach based on information retrieval, adapting document/query matching techniques to test case prioritization. Linear regression is used to determine relative weights of standard coverage information and guide information retrieval. Our results indicate the proposed approaches are generally more effective, and never worse than, existing baseline approaches, with improvements in fault detection rates up to 4.7% (with statistical significance).

The contributions of this work are as follows. First, to the best of our knowledge, the proposed approach is the first to apply the TF-IDF concept in information retrieval to test case prioritization. Second, our approach is the first to use linear regression to automatically weight the importance of each feature regarding fault finding, allowing us to supplant traditional coverage-based prioritization. Finally, we have conducted an empirical study of our approach against traditional coverage-based prioritization methods, demonstrating that it results in consistent improvements in fault detection rate.

This work represents a first step towards improved TCP approaches that better consider the frequency at which code is tested. In future work, the approach may be improved by using advanced IR techniques to find test cases that are closely related to code changes. Regarding regression models, more features can be added to the current regression model. Adopting a non-linear model may also improve the effectiveness of test case prioritization.

Acknowledgment

This work was supported by the Dual Use Technology Program (UM13018RD1).

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[2] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness." in *Proc. of the International Conference on Software Engineering*, 2014, pp. 435–445.

[3] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Fundamental Approaches to Software Engineering*, 2012, pp. 409–424.

[4] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *Proc. of the International Conference on Software Testing, Verification and Validation*, 2013, pp. 312–321.

[5] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *Proc. of the International Conference on Software Maintenance*, 2006, pp. 123–133.

[6] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Proc. of the International Symposium on Software Reliability Engineering*, 2003, pp. 442–453.

[7] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[8] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[9] S. Mirarab and L. Tahvildari, "An Empirical Study on Bayesian Network-based Approach for Test Case Prioritization," in *Proc. of the International Conference on Software Testing, Verification, and Validation*, 2008, pp. 278–287.

[10] G. Salton, *The SMART retrieval system – experiments in automatic document processing*. Prentice-Hall, Inc., 1971.

[11] C. M. Bishop and N. M. Nasrabadi, *Pattern Recognition and Machine Learning*. Springer New York, 2006, vol. 1.

[12] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[13] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, "Dodona: Automated oracle data set selection," in *Proc. of the International Symposium on Software Testing and Analysis*, 2014.

[14] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proc. of the International Conference on Automated Software Engineering*, 2011, pp. 612–615.

[15] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis," *Empirical Software Engineering*, vol. 11, no. 1, pp. 33–70, 2006.

[16] M. Staats, P. Loyola, and G. Rothermel, "Oracle-centric test case prioritization," in *Proc. of the International Symposium on Software Reliability Engineering*, 2012, pp. 311–320.

[17] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proc. of the International Conference on Software Engineering*, 2001, pp. 329–338.

[18] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of the International Conference on Software Engineering*, 2005, pp. 402–411.

[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*, vol. 32, no. 9, pp. 733–752, 2006.

[20] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *Proc. of the Computer Software and Applications Conference*, vol. 1, 2006, pp. 411–420.

[21] C. D. Nguyen, A. Marchetto, and P. Tonella, "Test case prioritization for audit testing of evolving web services using information retrieval techniques," in *Proc. of the International Conference on Web Services*, 2011, pp. 636–643.

[22] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.