EMPIRICALLY REVISITING AND ENHANCING IR-BASED TEST-CASE PRIORITIZATION

BY

QIANYANG PENG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Advisers:

Professor Darko Marinov
Assistant Professor Lingming Zhang, The University of Texas at Dallas

ABSTRACT

Regression testing is widely used to check code changes during software evolution. However, regression testing can also incur huge cost since real-world software systems can accumulate huge and time-consuming test suites. Therefore, a large number of approaches have been proposed to speed up regression testing, including test-case prioritization, regression test selection, and test-suite minimization/reduction.

Test-case prioritization (TCP) aims to detect regression bugs faster via reordering regression tests. To date, various TCP techniques have been proposed in the literature, including both *change-aware* ones that consider program changes between the old and new versions for better prioritization and *change-unaware* ones that simply use the dynamic or static information from the old version. Although various studies have investigated the effectiveness of the existing TCP techniques, they suffer from the following threats: (1) they are usually evaluated on a small dataset; (2) they are usually performed on seeded artificial or real bugs, and not on real evolution with real bugs; (3) they are usually evaluated using cost-unaware metrics.

In this work, we study the recent program changes based information retrieval (IR) approach for TCP, which has been claimed to perform better than state-of-the-art coverage based techniques but only evaluated on a small dataset by cost-unaware metrics. We reduce the threats of and further enhance the prior work by conducting a better evaluation and proposing potential improvements. To do so, we evaluate the original technique on a large-scale, real-world software-evolution dataset containing 123 projects and 2,980 program builds using both cost-aware and cost-unaware metrics under various configurations, and we design and evaluate several hybrid techniques combining the IR features and historical test execution times and failure frequencies.

As a result, we confirm the effectiveness of program changes based IR approach for TCP and find an ideal configuration to maximize the effectiveness. Also, we successfully improve the performance of the original technique with our hybrid techniques. Moreover, we show that flaky tests have a substantial impact on the program changes based TCP techniques.

To my parents and my fiancée, for their love and support.

ACKNOWLEDGMENTS

I would like to first thank my adviser Professor Darko Marinov. He encouraged and supported me to discover the research topic I am interested in so that during this year I could consistently work on what I really love. He kept guiding and encouraging me through my whole research process. I will never forget the late nights he and his student, August Shi, spent with me working on improving my research skills. He also has a very broad knowledge and quickly answered any tricky questions I had and solved many problems I ran into, thus greatly boosting the efficiency of my research work.

I would like to thank my co-adviser Assistant Professor Lingming Zhang from The University of Texas at Dallas. His area of study exactly matches my research interest, and he therefore gave me precious advices about how to turn my interest into a research work that could contribute the most to the research area. He also helped a lot with the feedback on this thesis.

I would like to thank the organizations that gave financial supports to me and the people who helped me with my thesis. My research assistantships were funded by the research grant from Huawei. My teaching assistantships were funded by University of Illinois Urbana-Champaign. NSF grants paid for the people who helped me with my work, including CCF-1421503, CCF-1763788, and CCF-1763906.

I would also like to express my gratitude to one of Professor Marinov's senior Ph.D. students, August Shi. He offered a lot of help including helping me to form my research idea, reviewing and revising my thesis, and answering my detailed technical questions.

Finally, I would like to thank Professor Marinov's Ph.D. students, Wing Lam and Owolabi Legunsen. They gave me very helpful suggestions regarding my thesis work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Regression testing is widely used to check code changes during software evolution [1]. However, regression testing can also incur huge cost since real-world software systems can accumulate huge and time-consuming test suites. For example, the Google engineers have witnessed a quadratic increase on their daily regression testing costs due to the linear increase in both the test-suite execution time and the number of software revisions; Google reports running over 150 million test executions every day, consuming substantial machine resources [2]. Therefore, a large number of approaches have been proposed to speed up regression testing, including test-case prioritization [3, 4, 5], regression test selection [6, 7, 8, 9], and test-suite minimization/reduction [10, 11, 12]. Yoo and Harman presented a thorough survey of regression testing techniques [13].

Test-case prioritization (TCP) aims to detect regression bugs faster by reordering regression tests [14]. To date, various techniques have been proposed in the literature, including both *change-aware* and *change-unaware* techniques. Change-unaware techniques simply use the dynamic or static information from the old version to perform TCP. For example, the *total* technique simply sorts all the tests in the descending order of the number of their covered program elements (e.g., methods or statements), while improved *additional* technique sorts the execution order of tests based on the number of their covered elements that are uncovered by already prioritized tests [15]. In contrast, change-aware techniques consider program changes between the old and new versions for even better prioritization. For example, the recent change-aware technique based on information retrieval (IR) [4] reduces the problem of TCP into the traditional IR problem—the program changes between revisions are treated as the *query*, while the tests are treated as the *data objects*. Then, the tests that are textually more related to the program changes are executed earlier for faster bug detection.

Although various studies have investigated the effectiveness of the existing TCP techniques, they suffer from the following threats. First, they are usually performed on seeded artificial bugs or seeded real bugs, and not on real-world software evolution with real test failures. For example, Luo et al. [16] empirically revisited TCP techniques on both artificial bugs via mutation and real bugs from Defects4J [17]. However, even Defects4J bugs are not representative of real regression bugs since Defects4J bugs are isolated bugs while real regression bugs often come with other benign changes. Second, the existing TCP techniques are usually evaluated using cost-unaware metrics, such as the Average Percentage of Faults Detected (APFD) [14]. Even state-of-the-art IR-based change-aware TCP technique was also evaluated using only APFD [4, 18]. Last but not least, *flaky tests* have been demonstrated to be prevalent in practice [19], while to our knowledge none of the existing studies for TCP considered the impacts of flaky tests.

In this work, we focus on the *change-based, IR* TCP technique (hereinafter called the "**IR-based TCP technique**"). As is introduced above, it uses IR approach to calculate the textual similarities between the program changes and test cases, then prioritizes test cases by the descending order of the similarities. We focus on the this specific technique because it has been shown to outperform state-of-the-art change-unaware total and additional techniques [4], and is an lightweight static technique thus easy to be evaluated on a large dataset. In our new evaluation we aim to reduce the threats of the prior studies, and to further enhance the technique by discovering more configurations and hybridizing IR features with historical test execution times and failure frequencies to make it perform even better.

To perform the study, we constructed a large-scale, real-world, software-evolution dataset including 123 projects and 2,980 program builds with real test failures. Then, we evaluate the studied technique with various different configurations on our dataset using both cost-aware and cost-unaware metrics. After that, we propose and evaluate the hybrid techniques related to our focused technique. Lastly, we study the impacts of flaky tests.

In summary, this thesis makes the following contributions:

- *Dataset:* We constructed a real-world, large-scale, software-evolution dataset with 2,042 Travis builds, 2,980 program builds (a Travis build could consist of multiple program builds), and 6,618 real test failures from 123 open-source Java projects. For each program build, our dataset has the source code, program changes, and the test execution information (the pass/fail outcome and the test execution time), which suffices for evaluating TCP techniques.

- *Evaluation:* We evaluated a wide range of configurations of IR-based TCP technique on our dataset using both cost-unaware and cost-aware metrics. We are the first to evaluate IR-based TCP technique with cost-aware metric, and we are the first to evaluate hybrid techniques of IR. To the best of our knowledge, we are also the first to study the impacts of flaky tests on TCP.

- *Outcome:* Our study reveals various practical guidelines for future TCP studies, including: (1) it is important to have a large-scale dataset for studying TCP; (2) there could be a huge bias when using cost-unaware metric for TCP on real-world software evolution; and (3) flaky tests have a substantial impact on the change-based techniques.

Besides the guiding outcomes, we also found an ideal configuration for IR-based TCP technique, and discovered hybrid techniques based on IR that significantly outperform the original IR-based TCP technique.

CHAPTER 2: INFORMATION RETRIEVAL (IR) TECHNIQUES

IR-based TCP technique was first proposed and evaluated by Saha *et al.* [4]. As the main purpose of our thesis is to better evaluate and enhance their technique, here we introduce the technique they propose with more details. Typically, an IR-based technique implements an IR system, which aims to trace and recover specific information from stored data, usually based on the textual similarities. Saha *et al.* reduce a TCP problem into an IR problem by first ranking the test cases (classes or methods) by the similarity scores between each test case and the modified code, and then using the ranking of the scores as the executing order of test cases. The assumption is that the test case that have a higher textual similarity with the modified code are more related to the program changes, thus having a higher chance to reveal the regression bugs between code versions early.

2.1 DATA OBJECTS AND QUERY CONSTRUCTION

There are three key parts for an IR system: the collection of data objects, the query, and the retrieval model. In the change-based TCP, the data objects are constructed from the test cases and the query is constructed from the program changes between two code versions. In this section we discuss the data preprocessing techniques used to construct the data objects and the query.

In the prior work of Saha *et al.*, they discuss and evaluate several possible combinations for the configurations of the data objects and query construction, including whether to use structured documents, whether to do AST parsing, whether to use compact queries, and whether at method level or class level to construct the data objects. Their result shows that different configurations perform pretty closely, so we are not going to evaluate all their configurations in our work. Our evaluation is only at the class level, and we do not construct structured documents or compact queries.

To see whether our result is consistent with the prior work, our evaluation still considers the impact of doing AST parsing. Also, we add two more configuration evaluations including the impact of tokenization, and the impact of the context of change added into the query.

2.1.1 Construction of Data Objects

In our IR system, data objects are constructed from the string representation of test files, and the preprocessing step is to process the raw strings of the files into tokens. In natural language processing, token is a sequence of characters acted as a useful semantic unit for processing. In our case, for example, a variable could be a token. We consider four different approaches to process the test files.

The most naive approach we consider is to use the white space tokenizer that breaks text into tokens separated by any whitespace character. We denote this approach as *Low*. The advantage of this approach is that it is easy to implement and oblivious to the programming language that the code files are written in. However, the disadvantage of this approach may include: (1) the approach does not filter out meaningless terms for IR such as Java keywords (e.g. *if*, *else* and *return*), operators, numbers and open source licenses, and (2) this approach fails to detect the similarities between the variable names that are partially but not exactly the same, e.g. $setWeight$ and *getweight*.

A optimization for the naive *Low* approach is to use a special code tokenizer to tokenize the source code and we denote this approach as Low_{token}. Low_{token} improves upon *Low* by introducing a code tokenizer that (1) filters out all the numbers and operators, (2) segments the long variables by non-alphabetical characters in between and by the camel-case heuristics, and (3) turn all upper case letters to lower case letters. With the help of code tokenizer, Low_{token} does not have most defects of *Low*. However, it still does not filter out some meaningless terms for IR, such as Java keywords and open source licenses.

The third approach is based on the assumption that identifiers are particularly important terms for information retrieval because developers use their own natural language terms to name them [4]. Therefore, we build an abstract syntax tree (AST) from each code file and extract the identifiers from it. We denote this approach as $High$. After extracting the identifiers from the source code, we could also apply the same code tokenizer as Low_{token} to create finer grained tokens. We denote this approach, which is our last approach, as $High_{token}$.

2.1.2 Construction of Queries

For IR-based TCP, we construct the query using the program changes between two repository versions. We only consider the textual part of program changes and ignore any non-textual files, such as images and binary files.

Similar to the construction of data objects, we could also apply one of *Low*, Low_{token}, $High$, and $High_{token}$ as the preprocessing approach for the query. For each type of data object construction, we apply the same corresponding preprocessing approach, e.g., if using $High_{token}$ for data objects, we use $High_{token}$ for the query as well. Unfortunately, we cannot collect identifiers related to exact program changes directly when doing $High$ or $High_{token}$, because most code parsers can only parse structurally complete code, not partial snippets from the diff. Instead, we collect identifiers of code contexts with three steps. In the first step, we parse the code in the new version and record the line number of every identifier in the code. In the second step, we generate the unified diff file and record the changed line numbers in the new version of code. Lastly, we collect the identifiers that are at the line numbers flagged as modified in the diff file.

Another important part of constructing the query is whether to include the *context* of change or not. By context, we mean the lines of code around the exact changed lines between two code versions.

In one extreme case, the context can be the whole changed file. With the whole changed file, the query will include any potentially related information within the same file. However, using the whole file as context would generate a very long query. Furthermore, using the entire file can be imprecise by providing too much information not related to the actual changes.

In the other extreme case, the context could be empty, so the query is just the changed lines of code. The query in this case is the shortest and more related to the exact changes. However, without any context, there might be no enough information in the query and the IR technique would report low-quality ranking results. Specifically, such a short query may result in more frequent ties in the similarity score between the query and multiple data objects.

The alternative approach is to include a certain number of lines of context into the query, and hopefully it would absorb both the advantage of both the "whole file as context" and "no context" approach. In our evaluation, we evaluate the impact of including 1 line, 3 lines and 5 lines of context into the query.

2.2 RETRIEVAL MODELS

Retrieval model is the last key part of an IR system. It takes the data objects and the query as input, and generates a ranking to data objects as the output. Finally, the ranking of data objects will become the prioritized order of test cases. All retrieval models we use are bag-of-words models [20], which means the order of words in a data object does not matter. Bag-of-words models has two major advantages: (1) when multiple files are modified and need to be merged into one query, the order of the files do not matter because the result would be the same, and (2) it simplifies the code representation and reduces the dimensionality of data.

We evaluate four retrieval models: Tf-idf, BM25, LSI and LDA. The first two models are commonly used unsupervised IR models, and the last two models are two simple topic models. The prior work [4] introduced all these 4 models, but only the BM25 model is used in their evaluation. Here we evaluate all these 4 models because we want to see how they perform and compare exactly when doing IR-based TCP.

- **Tf-idf**

  Tf-idf [21] is a bag-of-words based text vectorization algorithm. Given the vector representations of the data objects, we do the data object ranking based on the vector distances between the data objects' vectors and the query. We use the TfidfVectorizer from scikit-learn [22] to implement the Tf-idf model used in our evaluation.

  The idea of Tf-idf model is to use a combination of term frequencies and inverse document frequencies to compute the weight of every term in the data objects. The more frequent words are given higher weight by the term frequency score, while more common words are given lower weights by the inverse document frequency score.

  The training phase of the Tf-idf model is to calculate the term weight of every word in the data objects. In the dataset each data object is represented by an array of size N, in which N is the number of different distinct terms in the whole dataset. The weight of term t_i will be stored as the ith element of the array of each data object.

  For a term t and a data object d, the weight of term t in data object d is defined as Equation 2.1:

  $$tf\text{-}idf(t,d) = tf(t,d) \times idf(t) \tag{2.1}$$

  In our evaluation, the term frequency tf is the count of the term t in data object d, while the inverse document frequency idf is computed as shown in Equation 2.2:

  $$idf(t) = log(\frac{1+n_d}{1+df(t)} + 1) \tag{2.2}$$

  in which n_d is the total number of data objects, and $df(t)$ is the number of data objects that contain term t.

  Finally, we normalize the tf-idf vector by the $l2$ norm, defined by Equation 2.3:

$$v_{norm} = \frac{v}{\sqrt{{v_1}^2 + {v_2}^2 + ... + {v_n}^2}} \tag{2.3}$$

In our evaluation, the similarity score between a data object vector and the query vector is computed by the cosine similarity defined by Equation 2.4:

$$s(\vec{d}, \vec{q}) = \sum_{i=1}^{N} tf(t_i, d) \times tf(t_i, q) \times idf(t_i)^2 \tag{2.4}$$

We do test-case prioritization by sorting the test classes in a descending order of their cosine similarity scores to the query, and this retrieval model is denoted as **Tf-idf**.

- **BM25**

  The BM25 [23] model, also known as Okapi BM25, is another successful retrieval model [24]. Compared to the Tf-idf model, it takes the data object lengths into consideration, such that shorter data objects are given higher rankings. Moreover, unlike the traditional Tf-idf model which requires us to compute the cosine similarity scores to do the ranking after feature extraction, BM25 itself is designed to be a ranking algorithm that aims to directly compute the scores to be used in ranking.

  In our evaluation, we use the gensim [25] implementation of BM25. Given the query q as a sequence of words $[t_1, t_2, \ldots, t_m]$ (t_i and t_j could be the same words but at different positions), the similarity score between a data object d and q is computed as Equation 2.5:

$$s(d, q) = \sum_{i=1}^{m} idf(t_i) \cdot \frac{tf(t_i, d) \cdot (k_1 + 1)}{tf(t_i, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avgdl})} \tag{2.5}$$

In this equation, $|d|$ is the length of data object d, while $avgdl$ is the average length of all data objects in the dataset. The inverse document frequency $idf(t)$ is computed as Equation 2.6:

$$idf(t) = log(\frac{n_d - df(t) + 0.5}{df(t) + 0.5}) \tag{2.6}$$

where n_d is the total number of data objects, and $df(t)$ is the number of data objects that contain term t.

In the equation, k_1 and b are two configurable parameters. k_1 is a positive tuning parameter that controls the data object term frequency scaling, and b controls the scaling of data object length. Usually, k_1 is chosen from range $[1.2, 2.0]$ and b is 0.75. In our evaluation, k_1 is 1.5, which is the default value provided by gensim.

We sort the test classes in a descending order of their BM25 scores, and this retrieval model is denoted as **BM25**.

- **LSI and LDA**

  LSI [26] and LDA [27] are two classic unsupervised bag-of-words topic models. As a topic model, the representation of a data object is a vector of topics rather than a vector of tokens. This mathematical embedding model will transfer the data objects from a very high dimensional vector space with one dimension per word into a vector space with a much lower dimension. With a vector of topics for each data object, we can calculate the similarity scores and rank the data objects.

  We use the gensim implementation of LSI and LDA, and we use cosine similarity to calculate vector similarities to compute the ranking. These two retrieval models are denoted as **LSI** and **LDA**.

CHAPTER 3: DATA COLLECTION

3.1 PROJECT SELECTION

To evaluate how various TCP techniques prioritize failed tests, we need a real, large, and inclusive dataset of projects that have real test failures. Our project collection phase consists of three steps.

In the first step, we collect a list of projects by querying the GitHub search API. We require the primary programming language of projects to be Java and the last updated date to be after 2018-10-01. We only choose projects with recent updates because we prefer active projects. In the query we rank projects by the number of stars given to the project because high-starred project are more likely to be of higher quality and are likely to be more popular and relevant to others. To bypass the GitHub limitation on the number of returned items in each single query, we segment our query by the project creation date from 2010 to 2019 and choose 1,000 projects from each query. In this step we collect 10,000 projects.

In the second step, we select projects that have a Travis build history publicly accessible from Travis CI, a continuous-integration service widely popular among GitHub projects [28, 29]. The Travis build history includes the logs for each Travis build and a link to the program changes on GitHub that triggered the build. We select only the projects that have a `.travis.yml` file in their root directory. Also we require the project to have the `active` field set to `True` in the project's corresponding Travis CI setup. We end up with 2,181 projects after this step.

In the last step, we select projects that have analyzable logs to identify the Fully Qualified Names (FQNs) and the execution times of all tests. We choose projects that build with Maven (as identified by having a `pom.xml` file in the root of the directory) because it produces a more verbose output than Ant and Gradle, two other popular Java build tools. However, even if a project builds with Maven, it is challenging to parse the test FQNs and times, e.g., when the tests are wrapped in test suites, the test output is silenced, or the build log itself is broken or lost. We choose projects that have more than 30% of the builds in their build history that are analyzable (how they are analyzed in described in section 3.3), thus selecting 123 projects for our evaluation.

3.2 BUILD COLLECTION

Travis CI enables developers to define a "Build Matrix" in the configuration file. In the build matrix developers can define a group of different language-and-environment combinations, and each language-and-environment combination will be executed separately as a *job*. The execution results of all jobs will be integrated together to generate the execution result of the *build*. In this thesis, we refer to "Travis job" as "**program build**" because without context "job" is a pretty vague term and "program build" makes this term more precise and easier to understand.

If Travis CI is enabled in a GitHub repository, it will monitor the actions in the repository and trigger an automatic Travis build whenever there is a GitHub Push or Pull Request. During every program build, the tests for the code are executed and any failure in test will cause the build to fail. After each Travis build, Travis CI generates a Travis build log and updates the build history for the project. We collect and process the Travis build log details in the build history for our TCP evaluation.

We mainly care about the following attributes in a Travis build log:

- *event_type*: The type of the event that triggers a Travis build. The event can be automatically triggered from a GitHub "push" or "pull_request", or manually triggered using corresponding Travis API.

- *state* and *previous_state*: *State* is the current status of a Travis build. For a Travis build that has finished, the *state* can be "passed", "failed" or "errored". If the *event_type* for the build is *push*, then the *previous_state* property is copied from the last build on the same branch. If the *event_type* of the build is *pull_request*, then the property is copied from the last build on the branch that the pull request is being merged into.

- *commit*: The value corresponding to this attribute is a dictionary containing several related attributes about the commit associated with the Travis build. The attributes needed in our analysis are the *SHA* and *compare_url*. *SHA* is a hash identifier of the commit, with which we can download the whole project zip at that specific commit from GitHub. *compare_url* is the url to the page on GitHub showing the program changes between the previous version and the current version.

- *jobs*: A Travis build contains one or multiple jobs (program builds). Jobs are executed in parallel and the *state* of a Travis build is decided by the *state* of the jobs in it altogether. We could download the detailed job logs from Travis server given the job ID, and we use the job logs to extract the test FQNs, test outcomes, and test execution times.

Example 3.1 shows what a common Travis build log looks like. From the log we know that this Travis build is from a repository called “airlift”, and is triggered from a GitHub pull request. The build is failed but the prior build right before it was passed, therefore regressions are likely to be introduced after the last program change. We could get a zip file of the repository snapshot at the build time using the SHA of the commit, and get the program changes between two versions using the compare_url of the commit. This Travis build contains exactly one job (program build), and we can use the @href url of it to retrieve more detailed information about the program build.

```
{
    "id": 196380882,
    "state": "failed",
    "event_type": "pull_request",
    "previous_state": "passed",
    ...
    "repository": {
        ...
        "name": "airlift",
        "slug": "airlift/airlift"
    },
    "commit": {
        ...
        "sha": "7b18a62e23d49e760a79925c30d460375153a8fb",
        "compare_url": "https://github.com/airlift/airlift/pull/493",
        "committed_at": "2017-01-29T18:12:18Z"
    },
    "jobs": [
        {
            "@type": "job",
            "@href": "/job/196380883",
            "@representation": "minimal",
            "id": 196380883
        }
    ],
    ...
}
```

Example 3.1: A Typical Travis Build Log

For each of the 123 projects, we collect the builds that satisfy the two requirements listed below:

1. The “previous_state” attribute should be “passed”, and the “state” attribute should be “failed”.

2. In the program changes list there should be at least one Java code file that was modified by querying the compare_url.

For each program build, we collect the following data for our experiment:

- Build Profile: The repository name, Travis build ID and job (program build) ID.

- A list of changed files between the two code versions. For each changed file, we have (1) the new content and (2) the difference between the new content and old content.

- The program build log file at the current version and at the prior version. In next section we will discuss how we use these files in detail.

3.3 LOG ANALYSIS

For each program build, we collect its log files both at the current version and at the prior version. By analyzing the log files we want to extract the list of tests including the test FQNs, test outcomes, and test execution times. We just need the log file at the current version to evalate the pure IR techniques. However, when we implement the hybrid techniques, we need the log file at the prior version to get the execution times as an estimation of test costs. It is more real and convincing than just using the execution times at the current version, which are impossible to retrieve in practice before test cases are executed.

In our work we do a class level TCP, so all the information we collect is at a class level. In Section 3.1 we have declared that all our projects are built with Maven, so we use the Maven log syntax to analyze the log files. To get a list of executed test classes, we match each line of the program build log and find the lines that match the pattern starting with “Running ” and followed by the FQN of the test class. We use the FQN of a test class to find the file for it, thus getting its contents from the zip snapshot. When there is a “$” separating the class FQN and the name of its inner class, we take the part before “$”, essentially merging each inner test class with its outer class.

For the execution times, we match the pattern “Time elapsed: (.*? s)” in the lines following the execution of each test, obtaining a float number in seconds representing the time spent on each test. To make the evaluation more realistic, we add a small overhead to the time of each test class as the time overhead to switch between test classes. To get an accurate estimation of this overhead, we created a experimental maven project and monitored on

Travis the time spent on switching between classes. As a result, we add 0.0058 seconds to the execution time of each test class.

Sometimes there is no time recorded for a test execution because of some special configuration or the log being incomplete. To keep our experiments precise, we only keep the program builds that have **all** their tests mapped to a time at the current version. Note that We do not require every test class at the previous version to have a time mainly because there could be newly added tests, so the prior test time for a test class could be $None$. In this case, we assign an after overhead time 0.0029 seconds as the prior time such test class. This time assignment is the general overhead divided by 2 and gives unseen test classes a higher prioriy in cost-cognizant techniques.

The extraction of test failures is challenging because of the format in which JUnit and Surefire print failures. We use an open-source tool TravisTorrent [30], which provides a group of log parsing tools for Maven, Gradle and Ant. Using the tool we parse the test method FQNs of failed tests from the logs, and we map each failed test method FQN to its test class. The principle of log analysis script is to match the string pattern starting with "Failed tests:" or "Tests in error:" in the build log to find the method names that failed or errored during the program build.

Instead of extending our method to make it work for all the cases, we just select the program builds that are fully analyzable and discard the program builds we cannot analyze. We find 2,980 well formatted analyzable program builds from 2,042 Travis builds across 123 projects. The number of Travis builds and the number of program builds for each project are shown in Table 3.1.

3.4 FLAKY TEST COLLECTION

Not all test failures are due to the latest program changes; some test failures are due to the so-called *flaky tests* that can non-deterministically pass or fail even for the same code under test [19]. However, this effect could be substantial, especially for program changes based TCP techniques. If a test failure is due to flakiness, the test may have nothing to do with the recent change, and a program changes based TCP technique may (rightfully) rank such a test lower. However, if purely measuring TCP effectiveness based on test failures, such a ranking could lead the TCP technique being considered worse than it should. To study this effect, we build a dataset that splits test failures into two groups: those definitely due to flaky tests and those likely due to real regressions introduced by program changes.

Identifying whether a test failure is due to a flaky test or not is challenging and would in the limit require extensive manual effort [31]. We use an automated approach by re-running the commits for the program builds with failed tests exactly six times and checking what tests failed. A test that passes and fails in different reruns of the same commit is flaky (by definition). A test that fails in all reruns is likely (but not definitely) a real failure indicating a regression introduced by the program changes.

From our set of 2,980 program builds, we select 252 program builds that contain exactly one test failure and the with a build time of less than five minutes. For each of these builds, we rerun on Travis with exactly the same configuration six times and record whether that single failure re-occurred or not. We find 29 builds with definitely flaky tests and 223 builds with likely real failures. Note that having a consistent failure in six reruns does not necessarily mean the test is non-flaky.

ID	Name	#Builds	#Jobs	ID	Name	#Builds	#Jobs
P01	apache/incubator-dubbo	50	82	**P02**	alibaba/fastjson	46	46
P03	alibaba/druid	5	5	**P04**	ctripcorp/apollo	31	31
P05	eclipse-vertx/vert.x	25	95	**P06**	perwendel/spark	9	9
P07	xetorthio/jedis	45	67	**P08**	apache/incubator-druid	48	93
P09	google/auto	11	13	**P10**	jhy/jsoup	9	25
P11	apache/rocketmq	44	49	**P12**	vipshop/vjtools	14	21
P13	hs-web/hsweb-framework	25	25	**P14**	weibocom/motan	24	24
P15	square/moshi	4	5	**P16**	google/closure-compiler	31	38
P17	elasticjob/elastic-job-lite	44	86	**P18**	google/error-prone	18	26
P19	joelittlejohn/jsonschema2pojo	7	9	**P20**	apache/zeppelin	30	37
P21	Angel-ML/angel	6	6	**P22**	rest-assured/rest-assured	7	7
P23	socketio/socket.io-client-java	4	4	**P24**	spring-projects/spring-security-oauth	15	27
P25	keycloak/keycloak	42	52	**P26**	abel533/Mapper	10	12
P27	aws/aws-sdk-java	42	56	**P28**	cglib/cglib	1	1
P29	apache/incubator-dubbo-spring-boot-project	9	20	**P30**	alipay/sofa-rpc	39	72
P31	apache/incubator-pinot	46	52	**P32**	JanusGraph/janusgraph	31	144
P33	FasterXML/jackson-databind	24	44	**P34**	qos-ch/logback	1	1
P35	tcurdt/jdeb	4	4	**P36**	pholser/junit-quickcheck	3	3
P37	FasterXML/jackson-dataformat-xml	4	7	**P38**	basho/riak-java-client	17	39
P39	fakereplace/fakereplace	2	2	**P40**	JSQLParser/JSqlParser	16	34
P41	AxonFramework/AxonFramework	48	48	**P42**	internetarchive/heritrix3	9	15
P43	DiUS/java-faker	12	16	**P44**	zeroturnaround/zt-zip	2	3
P45	spring-projects/spring-data-mongodb	38	41	**P46**	spring-projects/spring-data-redis	30	41
P47	mitreid-connect/OpenID-Connect-Java-Spring-Server	5	5	**P48**	resteasy/Resteasy	45	101
P49	killme2008/aviator	4	4	**P50**	floodlight/floodlight	20	20
P51	gresrun/jesque	5	9	**P52**	opensagres/xdocreport	6	6
P53	onelogin/java-saml	2	4	**P54**	spring-projects/spring-data-cassandra	21	40
P55	ModeShape/modeshape	7	7	**P56**	LiveRamp/hank	7	7
P57	dooApp/FXForm2	10	10	**P58**	ocpsoft/rewrite	9	42
P59	demoiselle/framework	6	6	**P60**	apache/commons-compress	2	4
P61	shrinkwrap/resolver	2	3	**P62**	searchbox-io/Jest	13	13
P63	scobal/seyren	6	6	**P64**	pf4j/pf4j	6	13
P65	ebean-orm/ebean	31	44	**P66**	winder/Universal-G-Code-Sender	11	11
P67	magefree/mage	43	43	**P68**	openmrs/openmrs-core	48	48
P69	SpigotMC/BungeeCord	1	1	**P70**	junkdog/artemis-odb	12	12
P71	jenkinsci/java-client-api	3	3	**P72**	lukas-krecan/JsonUnit	22	57
P73	mjiderhamn/classloader-leak-prevention	1	1	**P74**	doanduyhai/Achilles	3	3
P75	st-js/st-js	11	11	**P76**	RoaringBitmap/RoaringBitmap	20	44
P77	graphhopper/jsprit	7	20	**P78**	prometheus/client_java	21	21
P79	alexxiyang/shiro-redis	8	8	**P80**	ff4j/ff4j	9	9
P81	awslabs/amazon-kinesis-client	17	28	**P82**	protegeproject/protege	3	3
P83	redpen-cc/redpen	12	16	**P84**	wmixvideo/nfe	20	20
P85	yegor256/rultor	13	13	**P86**	codelibs/fess	25	45
P87	undera/jmeter-plugins	10	10	**P88**	flaxsearch/luwak	3	3
P89	jaeksoft/opensearchserver	6	6	**P90**	sismics/reader	7	7
P91	mp911de/logstash-gelf	3	6	**P92**	teamed/qulice	6	12
P93	RIPE-NCC/whois	5	5	**P94**	jcabi/jcabi-github	4	4
P95	rapidoid/rapidoid	15	15	**P96**	pippo-java/pippo	11	11
P97	vert-x3/vertx-web	6	8	**P98**	sakaiproject/sakai	39	39
P99	yandex-qatools/postgresql-embedded	2	2	**P100**	rickfast/consul-client	9	9
P101	HubSpot/jinjava	7	7	**P102**	davidmoten/rxjava-extras	8	12
P103	gchq/Gaffer	24	41	**P104**	orbit/orbit	5	5
P105	zendesk/maxwell	7	27	**P106**	bootique/bootique	5	5
P107	eclipse/paho.mqtt.java	20	20	**P108**	apache/systemml	6	6
P109	amzn/ion-java	5	11	**P110**	stanford-futuredata/macrobase	16	16
P111	debezium/debezium	43	61	**P112**	jtablesaw/tablesaw	34	34
P113	vipshop/Saturn	30	42	**P114**	networknt/light-4j	25	25
P115	twitter/GraphJet	13	13	**P116**	alibaba/jetcache	4	4
P117	RipMeApp/ripme	44	74	**P118**	apache/servicecomb-java-chassis	49	49
P119	rhwayfun/spring-boot-learning-examples	4	4	**P120**	apache/servicecomb-pack	24	35
P121	alipay/sofa-bolt	10	20	**P122**	getheimdall/heimdall	15	15
P123	zhang-rf/mybatis-boost	19	19				

Table 3.1: Number of Travis builds and number of jobs (program builds) for each project

CHAPTER 4: EXPERIMENT SETUP

4.1 PROBLEM FORMULATION

In this sections we summarize the information we collected, and introduce whether they are used to do TCP or just used to compute the evaluation metrics. Also, we introduce the symbols needed when defining the evaluation metrics.

Given a build with n test classes and p modified files, we collect the following information of the program build to construct and evaluate the IR-based TCP techniques:

1. Program changes, represented by the changed files, between the current program build and the previous one: $C_f = \{C_{f1}, C_{f2}, ..., C_{fp}\}$

2. Program changes, represented by the different lines in each changed file, between the current program build and the previous one: $C_d = \{C_{d1}, C_{d2}, ..., C_{dp}\}$

3. Test classes in the current program build: $T = \{T_1, T_2, ..., T_n\}$

4. Time spent on executing each test in the previous program build: $t'(T_i) = t'_i, t'_i \in \{float, None\}$

5. Number of historical failures for each test in the current program build: $hf(T_i) = hf_i, hf_i \in \{int\}$

6. Test execution outcome (failed or not) for each test class: $f(T_i) = f_i, f_i \in \{True, False\}$

7. Time spent on executing each test in the current build: $t(T_i) = t_i, t_i \in \{float\}$

Among the information collected, (1) (2) (3) are used to construct the query and data objects of IR-based TCP techniques, (4) is used to construct the cost-based techniques, (5) is used to construct the historical failure-based techniques, and (6) (7) are used to evaluate the techniques.

4.2 RESEARCH QUESTIONS

This study aims at answering the following research questions:

- **RQ1:** How do different information retrieval configurations impact IR-based TCP techniques in real software evolution?

- **RQ2:** How do IR-based TCP techniques perform on different builds and on different projects when measured by cost-unaware and cost-aware metrics?
- **RQ3:** How can we further enhance IR-based TCP techniques?
- **RQ4:** How do different failure-to-Fault mappings impact the evaluation of TCP?
- **RQ5:** How do flaky tests impact TCP in real software evolution?

4.3 EVALUATION METRICS

We use two metrics Average Percentage Faults Detected (APFD) and Average Percentage of Fault Detected per Cost (APFDc) to evaluate the TCP effectiveness.

4.3.1 Average Percentage Faults Detected (APFD)

Average Percentage Faults Detected (APFD) [14] is a widely used cost-unaware metric to measure TCP effectiveness:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{4.1}$$

In the formula, n is the number of test cases and m is the number of failed tests. TF_i is the ranking of the ith failed test in the prioritized test suite.

4.3.2 Average Percentage of Fault Detected per Cost (APFDc)

Average Percentage of Fault Detected per Cost (APFDc) is a variant of APFD that aims to take into consideration the different fault severities and the costs of test executions [32, 33]. Usually it is difficult to retrieve the fault severities and the accurate cost, so typically researchers use a simplified version of APFDc that only takes into consideration the test execution time [34, 35]. Our evaluation utilizes this simplified version of APFDc in our evaluation:

$$APFDc = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j \times m} \tag{4.2}$$

In the formula, n, m, and TF_i have the same meanings as in the formula for APFD, while t_j represents the execution time of the jth test.

4.4 FAILURE-TO-FAULT MAPPING

Note that in prior work APFD and APFDc are defined with respect to the number of *faults* as opposed to failed tests. Prior work was evaluated using seeded faults and mutants, with an exact mapping from test failures to faults in the code. However, in our evaluation, we only have information concerning which tests failed, and we do not have an exact mapping from failures to faults. Potentially, a single failed test can map to multiple faults in the code, and conversely multiple failed tests can all be mapped to a single fault.

In our evaluation, we first assume that each single failed test maps to a single distinct fault. Unless explicitly declared, our evaluation will use these metrics with that assumption. In Section 5.4, we conduct most of our experiments again with a new mapping that all failures maps to the same fault, in order to see whether our conclusions are generalizable to different failure-to-Fault mappings.

4.5 TIE BREAKING

When multiple test classes share the same score or award value, this means the TCP technique considers them equally good. However, the tests still need to be ordered in some way. In our evaluation the equally good test classes are ordered deterministically in the order they appear in the program build log at the current version. That is to say, our TCP technique won't swap the order of two test classes if they are considered equally good by the technique, and the two test classes will be executed in the order as if no TCP technique is applied.

CHAPTER 5: EMPIRICAL EVALUATION

5.1 RQ1: IR CONFIGURATIONS

This RQ aims to find the best configuration combination for IR-based TCP technique; the best configuration will be utilized as the default implementation of the IR-based TCP technique for the later RQs. For the comparison between different configurations, we use APFD as the evaluation metric. The IR-based techniques are cost-unaware, so APFD would better reflect the ranking quality of different IR-based techniques.

As described in Section 2, there are three main parts we need to determine when implementing an IR-based TCP technique: (1) the preprocessing for data objects and queries, (2) the context of the program changes to construct the query, and (3) the retrieval model. These three parts are evaluated in RQ1.1, 1.2 and 1.3, respectively.

5.1.1 RQ1.1: Code Preprocessing

As discussed in section 2.1, there are four different approaches in terms of code preprocessing: Low, Low_{token}, $High$, and $High_{token}$. We apply these four preprocessing approaches to the test classes and program changes of our dataset to see the prioritization results. While the code preprocessing approach is the variable, we set the other variables as invariants. That is, the retrieval model is set as BM25, which is the retrieval model utilized in prior work [4], and the context of change is set as the whole changed file.

Method	**APFD**		**Time on IR**		**Avg. v-size**	
	mean	median	mean	median	data	query
Low	0.696	0.757	1.294	0.127	10419	3309
Low_{token}	0.755	0.849	2.194	0.326	1753	928
$High$	0.737	0.831	0.198	0.044	3056	677
$High_{token}$	0.752	0.844	0.558	0.160	1011	358

Table 5.1: Comparing different preprocessing approaches

The evaluation result is shown in Table 5.1.1. The table shows the information retrieval effectiveness represented in APFD, the time spent on running the information retrieval phase, and the mean vocabulary size representing the number of distinct terms in the data objects and in the query. We highlight the best configuration in gray for each metric, so the highlighted boxes have either the highest mean or median APFD, the shortest time, or the shortest vocabulary length.

From the table we have the following observations. First, code tokenization is an effective approach to reduce the vocabulary length. To illustrate, in the data objects, tokenization reduces the vocabulary size of raw source code by 83.2% and reduces the vocabulary size of extracted identifiers by 66.9%. Second, the configurations utilizing code tokenization usually take longer to run because the time complexity of the BM25 technique is linear to the number of words in the query (Section 2.2). That is, although code tokenization will reduce the vocabulary size, it will make the data objects and query longer by splitting long variable names into shorter terms. Third, doing AST parsing makes the information retrieval technique more time and space efficient. Finally, Low_{token}, $High$, and $High_{token}$ lead to similar APFD scores, while *Low* has clear inferior results than the other three configurations.

We use the $High_{token}$ configuration as our default data preprocessing strategy in the following evaluation, as it is just slightly (at most 0.005) worse than the best configuration for APFD, but is the most time and space efficient strategy among the four.

5.1.2 RQ1.2: Context of Change

The second variable that we can control when implementing IR techniques is how many lines of the context of changes should be included into the query. We describe how we get this controlled in Section 2.1.2.

One interesting observation is that when the changes between the previous and current commit is small, the IR scores for data objects tend to be identical (usually 0), so the prioritization barely changes the ordering of the tests from the original order. Such similarity ties are very common when we include no context of code into the query. Ideally, the configuration concerning context should lead to high APFD scores with a low rate of ties.

Context	**APFD**		**# Ties**	
	mean	median	mean	median
0 line	0.697	0.788	35.3	2
1 line	0.703	0.788	30.9	2
3 lines	0.713	0.799	25.8	1
5 lines	0.717	0.802	23.4	1
Whole-file	0.728	0.808	8.0	0

Table 5.2: Comparing different context of code

Table 5.1.2 shows the evaluation results about how different context information influences the APFD and the number of ties. Prior work [4] uses 0 line of context to construct their queries without evaluating the impact of it, but interestingly we find it actually performs the worst. Instead, using the whole file has the dominantly best performance in terms of both the highest APFD and the lowest rate of ties. Also, the less context of change the query has, the worse the performance of the IR system is. Therefore, we select the whole-file strategy as the default strategy to construct the queries.

5.1.3 RQ1.3: Retrieval Model

We introduce four different retrieval models in Section 2.2 and in this section we conduct the evaluation to compare these four models.

To choose an appropriate retrieval model, we do a general evaluation to the overall mean and median APFD value across all builds of each retrieval models. Besides that, we also perform a per-project evaluation to see for each retrieval model, the number of projects that perform the best (having the highest mean APFD) with it.

Retrieval Model	**APFD**		**# B.Projs**
	mean	median	
Tf-idf	0.728	0.812	27
BM25	0.752	0.844	75
LSI	0.707	0.775	13
LDA	0.659	0.735	8

Table 5.3: Comparing different retrieval models

Table 5.1.3 shows the comparison between the different retrieval models. Note that in this table and in the following tables, # **B.Projs** denotes the number of projects that has the highest mean metric value under each technique. We can see that the BM25 model has both the highest mean and median APFD, and over 60% of the projects in our evaluation have the best results when using BM25. Therefore, BM25 is chosen as the default retrieval model in the following evaluations.

In summary, we determine the best IR configuration is to use $High_{token}$ to preprocess data objects and query, use the whole changed file as the context for the query, and use the BM25 as the retrieval model. We use this configuration as our default configuration for our remaining evaluation, and we denote our IR-based TCP under this configuration as **TCPIR**.

5.2 RQ2: EVALUATION BY PROJECT

Our dataset contains 123 projects and 2,980 program builds with real failures and real execution times. With an evaluation by project with different metrics, we aim to answer three questions concerning TCP research:

1. How differently does the same TCP technique perform in different versions of the same project?

2. How differently does the same TCP technique perform in different projects?

3. Does do different TCP techniques compare when evaluated by different metrics?

In Figure 5.1, we show the results in a box plot showing the APFD and APFDc values for each build, using the TCPIR technique. For all the box plots in this thesis, the small blue number on the top of every box indicates the number of data points in the group, the triangle presents the mean value of the data points in the group, and the dash line in the middle of each box presents the median value of the data points in the group.

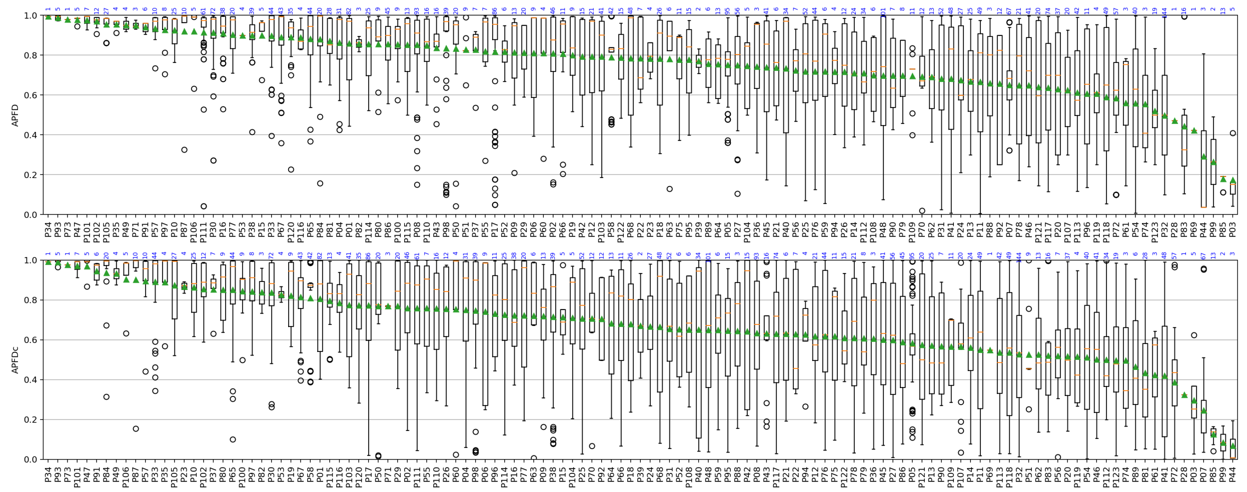

Figure 5.1: IR technique evaluation by each project

From the figure we have the following observations. First, the evaluation results for different version pairs of each project can differ significantly in terms of all the studied metrics. For example, the APFD values for project *apache/incubator-dubbo* range from 0.444 to 0.998. That is to say, evaluating IR-based TCP technique only on a small number of version pairs for each project could be easily biased. Second, the evaluation result also differs a lot across different projects. For example, among projects with more than 20 program builds, the mean

APFDc value ranges from 0.246 (*xetorthio/jedis*) to 0.892 (*FasterXML/jackson-databind*). That is to say, evaluating IR-based TCP technique on a small number of projects could also be easily biased. Third, when evaluating IR-based TCP technique, the projects with high APFD values also tend to have high APFDc values. However, for most projects, their APFD values are usually higher than the corresponding APFDc values. The reason is that the IR-based TCP technique only considers the textual relevance information of the tests and does not consider the test execution times.

The third question proposed at the concerns the impact of using different metrics in the evaluation of different TCP techniques. Prior TCP work has shown that the classic TCP evaluation metric APFD could be very biased and unreal. To be specific, a technique with high APFD does not necessarily have a high APFDc, thus being not cost-efficient [33, 34]. Chen *et al.* [34] showed that this simple cost-only technique outperforms three popular TCP techniques in academia including coverage-based greedy prioritization [36, 37, 18, 38], search-based prioritization [36, 37] and adaptive random prioritization [39]. However, there has not been prior work on using cost-aware metrics to evalute change-based IR techniques.

We compare a change-based IR-based TCP technique with a simple cost-only TCP technique that prioritizes test classes by the ascending order of their execution times in the prior program build. Ideally, the IR-based TCP technique should perform better than this simple cost-only technique; otherwise, the IR-based TCP techniques would not be a good choice to use. In our following evaluation, the cost-only TCP technique will be frequently used in the comparisons. As this technique runs quickest test first, we denote this technique as **QTF**.

Table 5.4 shows the results of the comparison between the TCPIR technique with the QTF technique in terms of the cost-unaware and cost-aware metrics. From the table we could have the following observations:

1. The cost-unaware metric APFD can severely over-estimate the performance of cost-unaware techniques and under-estimate the performance of cost-aware techniques. In our evaluation, if we take the mean APFDc value as the baseline, the mean APFD over-estimated the performance of TCPIR by 12.9% and under-estimated the performance of QTF by 50.0%. These results suggest that APFD is biased and should not be used to evaluate TCP techniques, where cost is an important factor.

2. TCPIR is better than QTF for 49.6% of the projects when evaluated by APFDc. This percentage is much higher than the percentage 15% reported by prior work [34] when comparing other state-of-the-art techniques with the cost-only technique. However, our IR-based TCP is still not better than the naive cost-only technique.

Avg. APFD								
ID	TCPIR	QTF	**ID**	TCPIR	QTF	**ID**	TCPIR	QTF
P01	0.860	0.351	**P02**	0.809	0.479	**P03**	0.174	0.206
P04	0.863	0.366	**P05**	0.752	0.314	**P06**	0.811	0.478
P07	0.650	0.401	**P08**	0.852	0.233	**P09**	0.814	0.174
P10	0.926	0.324	**P11**	0.667	0.259	**P12**	0.794	0.623
P13	0.669	0.488	**P14**	0.710	0.256	**P15**	0.900	0.197
P16	0.909	0.268	**P17**	0.819	0.350	**P18**	0.783	0.746
P19	0.800	0.303	**P20**	0.630	0.161	**P21**	0.738	0.431
P22	0.784	0.256	**P23**	0.784	0.170	**P24**	0.675	0.211
P25	0.719	0.501	**P26**	0.717	0.270	**P27**	0.750	0.333
P28	0.472	0.314	**P29**	0.812	0.502	**P30**	0.914	0.431
P31	0.682	0.408	**P32**	0.498	0.427	**P33**	0.898	0.509
P34	0.995	0.053	**P35**	0.956	0.485	**P36**	0.744	0.059
P37	0.829	0.631	**P38**	0.900	0.218	**P39**	0.770	0.145
P40	0.734	0.237	**P41**	0.682	0.117	**P42**	0.794	0.139
P43	0.837	0.734	**P44**	0.293	0.963	**P45**	0.739	0.127
P46	0.648	0.349	**P47**	0.978	0.424	**P48**	0.699	0.584
P49	0.952	0.383	**P50**	0.834	0.093	**P51**	0.829	0.143
P52	0.818	0.287	**P53**	0.900	0.100	**P54**	0.559	0.160
P55	0.820	0.117	**P56**	0.724	0.323	**P57**	0.935	0.592
P58	0.791	0.129	**P59**	0.719	0.268	**P60**	0.811	0.806
P61	0.561	0.220	**P62**	0.690	0.308	**P63**	0.781	0.439
P64	0.755	0.479	**P65**	0.883	0.477	**P66**	0.807	0.185
P67	0.892	0.165	**P68**	0.786	0.444	**P69**	0.423	0.500
P70	0.693	0.721	**P71**	0.951	0.034	**P72**	0.584	0.662
P73	0.980	0.260	**P74**	0.555	0.194	**P75**	0.778	0.296
P76	0.719	0.573	**P77**	0.903	0.442	**P78**	0.649	0.146
P79	0.698	0.469	**P80**	0.857	0.800	**P81**	0.871	0.248
P82	0.859	0.612	**P83**	0.444	0.091	**P84**	0.880	0.263
P85	0.179	0.175	**P86**	0.856	0.160	**P87**	0.921	0.396
P88	0.659	0.961	**P89**	0.758	0.318	**P90**	0.698	0.040
P91	0.941	0.792	**P92**	0.658	0.275	**P93**	0.989	0.555
P94	0.718	0.076	**P95**	0.778	0.324	**P96**	0.608	0.241
P97	0.927	0.217	**P98**	0.835	0.171	**P99**	0.265	0.689
P100	0.854	0.351	**P101**	0.976	0.259	**P102**	0.971	0.047
P103	0.793	0.484	**P104**	0.747	0.536	**P105**	0.956	0.581
P106	0.920	0.639	**P107**	0.624	0.423	**P108**	0.700	0.345
P109	0.696	0.318	**P110**	0.850	0.289	**P111**	0.916	0.210
P112	0.710	0.405	**P113**	0.612	0.229	**P114**	0.857	0.240
P115	0.854	0.247	**P116**	0.889	0.173	**P117**	0.638	0.319
P118	0.589	0.113	**P119**	0.607	0.720	**P120**	0.891	0.127
P121	0.641	0.218	**P122**	0.788	0.203	**P123**	0.520	0.537
#B.P	114	9	**Mean**	0.759	0.348	**Median**	0.784	0.314

Avg. APFDc								
ID	TCPIR	QTF	**ID**	TCPIR	QTF	**ID**	TCPIR	QTF
P01	0.806	0.921	**P02**	0.761	0.827	**P03**	0.296	0.828
P04	0.752	0.730	**P05**	0.584	0.887	**P06**	0.742	0.791
P07	0.246	0.867	**P08**	0.635	0.661	**P09**	0.721	0.498
P10	0.865	0.651	**P11**	0.550	0.566	**P12**	0.620	0.688
P13	0.572	0.723	**P14**	0.561	0.780	**P15**	0.715	0.518
P16	0.724	0.677	**P17**	0.774	0.819	**P18**	0.680	0.847
P19	0.822	0.748	**P20**	0.518	0.388	**P21**	0.631	0.721
P22	0.628	0.692	**P23**	0.868	0.836	**P24**	0.668	0.548
P25	0.708	0.665	**P26**	0.757	0.750	**P27**	0.600	0.676
P28	0.325	0.778	**P29**	0.770	0.837	**P30**	0.839	0.676
P31	0.655	0.500	**P32**	0.529	0.687	**P33**	0.892	0.840
P34	0.992	0.563	**P35**	0.890	0.838	**P36**	0.606	0.530
P37	0.854	0.800	**P38**	0.716	0.793	**P39**	0.671	0.762
P40	0.651	0.861	**P41**	0.421	0.593	**P42**	0.643	0.943
P43	0.632	0.897	**P44**	0.069	0.994	**P45**	0.601	0.678
P46	0.503	0.768	**P47**	0.970	0.899	**P48**	0.650	0.843
P49	0.935	0.890	**P50**	0.771	0.675	**P51**	0.526	0.794
P52	0.653	0.531	**P53**	0.828	0.609	**P54**	0.513	0.472
P55	0.760	0.732	**P56**	0.519	0.865	**P57**	0.896	0.864
P58	0.811	0.463	**P59**	0.649	0.521	**P60**	0.754	0.969
P61	0.423	0.515	**P62**	0.526	0.594	**P63**	0.721	0.624
P64	0.683	0.829	**P65**	0.851	0.823	**P66**	0.682	0.626
P67	0.815	0.429	**P68**	0.667	0.790	**P69**	0.549	0.928
P70	0.708	0.629	**P71**	0.770	0.227	**P72**	0.388	0.857
P73	0.976	0.422	**P74**	0.497	0.636	**P75**	0.618	0.732
P76	0.620	0.840	**P77**	0.723	0.866	**P78**	0.610	0.775
P79	0.608	0.729	**P80**	0.852	0.840	**P81**	0.435	0.803
P82	0.840	0.830	**P83**	0.521	0.530	**P84**	0.937	0.698
P85	0.127	0.563	**P86**	0.589	0.551	**P87**	0.902	0.899
P88	0.645	0.990	**P89**	0.466	0.617	**P90**	0.568	0.222
P91	0.945	0.952	**P92**	0.706	0.629	**P93**	0.991	0.893
P94	0.627	0.188	**P95**	0.646	0.721	**P96**	0.733	0.628
P97	0.844	0.873	**P98**	0.748	0.465	**P99**	0.085	0.966
P100	0.845	0.544	**P101**	0.971	0.307	**P102**	0.856	0.429
P103	0.776	0.489	**P104**	0.712	0.794	**P105**	0.874	0.857
P106	0.904	0.748	**P107**	0.566	0.807	**P108**	0.652	0.868
P109	0.567	0.860	**P110**	0.759	0.574	**P111**	0.760	0.660
P112	0.501	0.764	**P113**	0.537	0.494	**P114**	0.726	0.572
P115	0.797	0.651	**P116**	0.785	0.770	**P117**	0.631	0.729
P118	0.537	0.384	**P119**	0.516	0.757	**P120**	0.774	0.280
P121	0.575	0.411	**P122**	0.611	0.623	**P123**	0.498	0.778
#B.P	61	62	**Mean**	0.672	0.696	**Median**	0.671	0.729

Table 5.4: Comparing IR-based technique and quickest test first technique (#B.P = # B.Projs)

5.3 RQ3: HYBRID TECHNIQUES

5.3.1 RQ3.1: Hybrid Technique Considering Cost

In previous research questions, we derive the conclusion that IR-based TCP technique is indeed cost-aware effective in doing TCP tasks. Its performance is close to cost-only TCP technique and is better than some other state-of-the-art cost-unaware TCP techniques.

Prior work [34, 33] has tried to address the cost-inefficiency of traditional coverage based TCP techniques by implementing hybrid techniques that balance the coverages and the test times. However, these techniques perform similarly or even worse than the cost-only technique. Chen *et al.* [34] showed that for only 54.0% of their studied projects their hybrid technique performed better than cost-only technique, and the mean APFDc of hybrid technique is just 3% better than cost-only technique. The possible reason could be that traditional TCP techniques tend to put more costly test cases earlier, and this feature is negatively affecting each other with the cost cognizant feature which tend to put less costly test cases earlier.

Avg. APFDc

ID	QTF	CCIR	ID	QTF	CCIR	ID	QTF	CCIR	ID	QTF	CCIR	ID	QTF	CCIR	ID	QTF	CCIR
P01	0.921	0.944	**P02**	0.827	0.898	**P03**	0.828	0.783	**P04**	0.730	0.771	**P05**	0.887	0.884	**P06**	0.791	0.757
P07	0.867	0.854	**P08**	0.661	0.701	**P09**	0.498	0.593	**P10**	0.651	0.740	**P11**	0.566	0.610	**P12**	0.688	0.701
P13	0.723	0.740	**P14**	0.780	0.796	**P15**	0.518	0.625	**P16**	0.677	0.777	**P17**	0.819	0.844	**P18**	0.847	0.871
P19	0.748	0.861	**P20**	0.388	0.407	**P21**	0.721	0.756	**P22**	0.692	0.754	**P23**	0.836	0.844	**P24**	0.548	0.645
P25	0.665	0.746	**P26**	0.750	0.802	**P27**	0.676	0.732	**P28**	0.778	0.787	**P29**	0.837	0.851	**P30**	0.676	0.813
P31	0.500	0.696	**P32**	0.687	0.662	**P33**	0.840	0.899	**P34**	0.563	0.719	**P35**	0.838	0.853	**P36**	0.530	0.579
P37	0.800	0.849	**P38**	0.793	0.843	**P39**	0.762	0.738	**P40**	0.861	0.844	**P41**	0.593	0.594	**P42**	0.943	0.945
P43	0.897	0.908	**P44**	0.994	0.994	**P45**	0.678	0.649	**P46**	0.768	0.735	**P47**	0.899	0.950	**P48**	0.843	0.879
P49	0.890	0.899	**P50**	0.675	0.708	**P51**	0.794	0.869	**P52**	0.531	0.481	**P53**	0.609	0.609	**P54**	0.472	0.420
P55	0.732	0.736	**P56**	0.865	0.857	**P57**	0.864	0.888	**P58**	0.463	0.564	**P59**	0.521	0.682	**P60**	0.969	0.956
P61	0.515	0.513	**P62**	0.594	0.561	**P63**	0.624	0.782	**P64**	0.829	0.827	**P65**	0.823	0.900	**P66**	0.626	0.668
P67	0.429	0.438	**P68**	0.790	0.858	**P69**	0.928	0.831	**P70**	0.629	0.783	**P71**	0.227	0.320	**P72**	0.857	0.863
P73	0.422	0.941	**P74**	0.636	0.606	**P75**	0.732	0.727	**P76**	0.840	0.847	**P77**	0.866	0.916	**P78**	0.775	0.776
P79	0.729	0.729	**P80**	0.840	0.863	**P81**	0.803	0.744	**P82**	0.830	0.885	**P83**	0.530	0.486	**P84**	0.698	0.934
P85	0.563	0.578	**P86**	0.551	0.568	**P87**	0.899	0.949	**P88**	0.990	0.993	**P89**	0.617	0.540	**P90**	0.222	0.154
P91	0.952	0.963	**P92**	0.629	0.680	**P93**	0.893	0.985	**P94**	0.188	0.333	**P95**	0.721	0.730	**P96**	0.628	0.662
P97	0.873	0.912	**P98**	0.465	0.524	**P99**	0.966	0.966	**P100**	0.544	0.636	**P101**	0.307	0.860	**P102**	0.429	0.563
P103	0.489	0.618	**P104**	0.794	0.858	**P105**	0.857	0.863	**P106**	0.748	0.820	**P107**	0.807	0.809	**P108**	0.868	0.931
P109	0.860	0.875	**P110**	0.574	0.779	**P111**	0.660	0.784	**P112**	0.764	0.777	**P113**	0.494	0.514	**P114**	0.572	0.640
P115	0.651	0.775	**P116**	0.770	0.746	**P117**	0.729	0.731	**P118**	0.384	0.434	**P119**	0.757	0.904	**P120**	0.280	0.571
P121	0.411	0.392	**P122**	0.623	0.681	**P123**	0.778	0.779									
# B.Projs	26	94	**Mean**	0.696	0.744	**Median**	0.729	0.776									

Table 5.5: Comparing cost-only technique and cost-cognizant IR technique

Avg. APFDc

ID	HIS	CCH	CCHIR	ID	HIS	CCH	CCHIR	ID	HIS	CCH	CCHIR	ID	HIS	CCH	CCHIR	ID	HIS	CCH	CCHIR
P01	0.566	0.935	0.950	**P02**	0.563	0.875	0.914	**P03**	0.825	0.921	0.818	**P04**	0.508	0.720	0.768	**P05**	0.912	0.938	0.933
P06	0.763	0.867	0.948	**P07**	0.843	0.911	0.915	**P08**	0.674	0.808	0.836	**P09**	0.676	0.616	0.701	**P10**	0.421	0.587	0.704
P11	0.669	0.668	0.683	**P12**	0.451	0.680	0.696	**P13**	0.582	0.832	0.834	**P14**	0.579	0.794	0.809	**P15**	0.329	0.483	0.625
P16	0.629	0.741	0.814	**P17**	0.396	0.836	0.858	**P18**	0.618	0.853	0.878	**P19**	0.398	0.753	0.865	**P20**	0.675	0.627	0.631
P21	0.740	0.860	0.878	**P22**	0.631	0.753	0.805	**P23**	0.776	0.842	0.875	**P24**	0.742	0.639	0.700	**P25**	0.734	0.757	0.802
P26	0.407	0.766	0.807	**P27**	0.553	0.712	0.777	**P28**	0.281	0.778	0.787	**P29**	0.494	0.822	0.795	**P30**	0.749	0.836	0.895
P31	0.614	0.689	0.740	**P32**	0.555	0.697	0.705	**P33**	0.563	0.850	0.908	**P34**	0.803	0.629	0.775	**P35**	0.632	0.847	0.853
P36	0.369	0.530	0.579	**P37**	0.122	0.788	0.838	**P38**	0.606	0.813	0.864	**P39**	0.113	0.775	0.753	**P40**	0.547	0.866	0.862
P41	0.700	0.736	0.705	**P42**	0.568	0.948	0.951	**P43**	0.761	0.904	0.915	**P44**	0.571	0.994	0.994	**P45**	0.691	0.750	0.750
P46	0.547	0.811	0.780	**P47**	0.710	0.912	0.957	**P48**	0.763	0.873	0.904	**P49**	0.454	0.891	0.904	**P50**	0.701	0.746	0.768
P51	0.588	0.821	0.892	**P52**	0.325	0.568	0.489	**P53**	0.479	0.609	0.627	**P54**	0.613	0.523	0.514	**P55**	0.233	0.742	0.742
P56	0.635	0.866	0.857	**P57**	0.529	0.874	0.888	**P58**	0.854	0.893	0.926	**P59**	0.343	0.535	0.682	**P60**	0.669	0.969	0.956
P61	0.364	0.515	0.513	**P62**	0.616	0.736	0.728	**P63**	0.670	0.801	0.814	**P64**	0.425	0.825	0.835	**P65**	0.603	0.864	0.924
P66	0.304	0.647	0.688	**P67**	0.781	0.483	0.514	**P68**	0.562	0.798	0.873	**P69**	0.207	0.928	0.831	**P70**	0.771	0.843	0.853
P71	0.432	0.338	0.431	**P72**	0.529	0.880	0.850	**P73**	0.080	0.422	0.941	**P74**	0.118	0.623	0.590	**P75**	0.671	0.801	0.793
P76	0.526	0.861	0.855	**P77**	0.490	0.880	0.925	**P78**	0.644	0.834	0.876	**P79**	0.381	0.582	0.610	**P80**	0.391	0.868	0.872
P81	0.262	0.769	0.764	**P82**	0.475	0.859	0.903	**P83**	0.663	0.694	0.571	**P84**	0.690	0.806	0.943	**P85**	0.995	0.925	0.870
P86	0.547	0.611	0.627	**P87**	0.557	0.921	0.957	**P88**	0.730	0.990	0.993	**P89**	0.319	0.519	0.540	**P90**	0.745	0.439	0.362
P91	0.565	0.941	0.961	**P92**	0.568	0.671	0.695	**P93**	0.215	0.893	0.985	**P94**	0.755	0.607	0.687	**P95**	0.539	0.775	0.781
P96	0.394	0.638	0.665	**P97**	0.536	0.903	0.926	**P98**	0.623	0.705	0.764	**P99**	0.035	0.966	0.966	**P100**	0.529	0.554	0.685
P101	0.693	0.587	0.907	**P102**	0.923	0.767	0.829	**P103**	0.706	0.706	0.783	**P104**	0.575	0.814	0.875	**P105**	0.502	0.850	0.844
P106	0.595	0.759	0.855	**P107**	0.729	0.813	0.815	**P108**	0.662	0.901	0.941	**P109**	0.419	0.879	0.888	**P110**	0.532	0.674	0.824
P111	0.556	0.776	0.820	**P112**	0.456	0.815	0.815	**P113**	0.606	0.632	0.645	**P114**	0.497	0.651	0.743	**P115**	0.720	0.747	0.811
P116	0.473	0.706	0.751	**P117**	0.576	0.777	0.777	**P118**	0.582	0.484	0.534	**P119**	0.110	0.757	0.904	**P120**	0.765	0.716	0.833
P121	0.866	0.669	0.613	**P122**	0.382	0.652	0.695	**P123**	0.310	0.768	0.768								
# B.P	12	23	86	**Mean**	0.556	0.758	0.794	**Median**	0.568	0.776	0.815								

Table 5.6: Comparing history-based technique, cost-cognizant history based technique, and three factor hybrid technique

The change-based IR techniques we focus on studying is totally based on the textual similarity and is not giving stronger preference to complicated thus costly tests. Therefore, we could expect that IR techniques could potentially contribute more positively to cost-only techniques.

In Table 5.5 we show the result of comparing the APFDc of two techniques: The cost-only technique, also known as the QTF technique introduced in RQ2. And, the cost-cognizant hybrid technique which computes the award value for each test class by dividing its IR similarity score ir_i to the code change by the its execution time t'_i in the prior build:

$$a_{ccir}(T_i) = ir_i/t'_i \tag{5.1}$$

We denote the hybrid technique as **CCIR** which means "Cost-cognizant Information Retrieval". According to Table 5.5 we have the following observations: Firstly, the cost-cognizant IR technique (hybrid technique) outperforms the cost-only technique for 76.4% percent of the projects, and this percentage is much higher than 54.0% reported by Chen. Secondly, the hybrid technique CCIR performs significantly better than both the single factor technique TCPIR (see RQ2, by 10.7%) and QTF (by 6.9%) from which this hybrid technique is constructed. In contrast, in Chen's work their cost-cognizant technique performs just 3.0% better than cost-only technique.

5.3.2 RQ3.2: Hybrid Technique Considering Cost and Historical Failures

Historical test failures is a important factor that both industry [40, 2] and academia [41, 42, 43] would usually consider when optimizing software testing. Their consideration is usually based on the hypothesis that more frequently failed or more recently failed test cases are more likely to fail again in the future. As we can easily get the historical test failure frequency from our dataset, in this section we consider further enhance the performance of our hybrid technique with historical failures introduced into it. That is, we evaluate three factor hybrid techniques that considers IR scores, prior test times, and historical test failure frequencies.

We collect the historical failure frequency data for each test class in each program build by counting the number of times that the specific test class has ever failed before the specific program build. We define three historical test execution outcome based techniques. First, we define a TCP technique that directly arranges the test classes by the descending order of the historical failure frequencies hf_i, and denote this approach as HIS.

Second, for a test class T_i, given its historical failure frequency hf_i and its prior execution time t'_i, we define a cost-cognizant history based TCP technique that rearranges tests by the descending order of:

$$a_{cch}(T_i) = hf_i/t'_i \tag{5.2}$$

We denote this technique as **CCH** and our expectation is to see that CCH should perform better than the history only technique HIS and cost-only technique QTF.

Last we define a three factor hybrid technique which introduces IR feature into the cost-cognizant history based TCP technique. That is, by using the IR score ir_i for each test class, we invent a hybrid TCP technique that rearranges tests by the descending order of:

$$a_{cchir}(T_i) = (hf_i * ir_i)/t'_i \tag{5.3}$$

We denote this hybrid technique as **CCHIR**. Our expectation is that this three factor could outperform all the other techniques evaluated in this research work.

The comparison between HIS, CCHand CCHIRevaluated by APFDc is shown in Table 5.6. First we observe that the history-only technique (HIS) does not have a very good performance in terms of APFDc, however the cost-cognizant history based technique (CCH) performs even better than cost-cognizant IR based technique (CCIR in Table 5.5). This indicates that history information is a very effective factor in doing TCP, however when the historical test failure frequency is not abundant it is more good choice to hybridizing it with prior times rather than utilizing it alone. Another interesting observation is that the three factor hybrid technique combining IR scores, prior execution time and test failure frequency outperforms any other technique we evaluate in this thesis by at least 4.7% in terms of the mean averaged APFDc, which means the three factors we consider in the hybrid technique could enhance each other and form a effective hybrid technique.

5.3.3 RQ3.3: Statistic Test

To see whether there is a significant difference between the distributions of different techniques, we perform a ANOVA test and a Tukey HSD test on the APFDc result on the three single techniques (TCPIR, HIS, QTF) and three hybrid techniques (CCIR, CCH, CCHIR) and the result is shown in Table 5.3.3.

Tech	Mean	Group
CCHIR	0.797	A
CCH	0.771	B
CCIR	0.739	C
QTF	0.697	D
TCPIR	0.653	E
HIS	0.611	F
p-value	$<$ 2e-16	

Table 5.7: Statistic tests on different techniques

The evaluation result is shown in Table 5.3.3. Notice that the mean value shown for each technique here is different from the evaluation results by projects because this is the program build level mean rather than a project level mean. The capital letters A-F is the result of Tukey HSD test and the p-value is the result of ANOVA test. The result shows that there is a significant different between all these 5 techniques, with CCHIR being the best and HIS being the worst. Now we are confident to say that our hybrid techniques, especially the three factor hybrid technique, is a more effective technique to perform TCP comparing to the other techniques.

5.4 RQ4: FAILURE-TO-FAULT MAPPING

All our evaluation so far is based on the assumption that each failure maps to a distinct fault. In this section we evaluate again the major research questions by mapping all failures to the same fault to see whether the conclusions still remain the same. Like the previous RQs, we calculate the mean and median APFD by program build for Tf-idf, TCPIR, LSI and LDA, the mean and median average APFD by project for TCPIR and QTF, and the mean and median average APFDc by project for TCPIR, QTF, CCIR, HIS, CCH, CCHIR with all failures mapping to the same fault. We compare the results with mapping each failures to a distinct fault to see if there is a significant difference.

RQ No.		*One to One*		*All to One*	
RQ1	**APFD**	**mean**	**median**	**mean**	**median**
	Tf-idf	0.728	0.812	0.771	0.885
	BM25	0.752	0.844	0.795	0.913
	LSI	0.707	0.775	0.753	0.860
	LDA	0.659	0.735	0.713	0.837
RQ2	**Avg. APFD**	**mean**	**median**	**mean**	**median**
	TCPIR	0.759	0.784	0.802	0.830
	QTF	0.348	0.314	0.398	0.364
RQ2&3	**Avg. APFDc**	**mean**	**median**	**mean**	**median**
	TCPIR	0.672	0.671	0.727	0.730
	QTF	0.696	0.729	0.733	0.778
	CCIR	0.744	0.776	0.779	0.804
	HIS	0.556	0.568	0.606	0.635
	CCH	0.758	0.776	0.793	0.826
	CCHIR	0.794	0.815	0.826	0.853

Table 5.8: Impact of different failure-to-Fault mappings

Table 5.4 shows the evaluation results. From the table, although the APFD and APFDc values are larger when mapping all failures to one fault than when mapping each failure to distinct faults, different failure-to-Faults mappings lead to similar overall conclusions.

5.5 RQ5: FLAKY TESTS

So far, our evaluation assumes that all test failures are deterministic and indicate faults introduced by the changes in the code, i.e., there are no flaky tests. In this section, we evaluate the effects of flaky tests using a dataset of 252 builds that contain exactly one test failure with its flakiness labeled (Section 3.4).

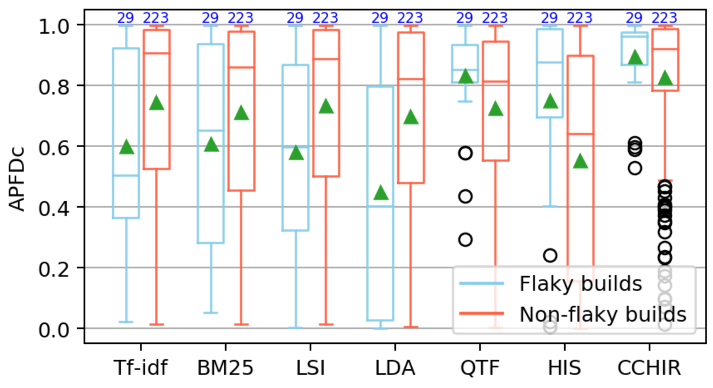

Figure 5.2: Comparing different techniques on flaky/non-flaky tests

For a build in our dataset, if the only failure it contains is flaky, then we call this build a *flaky build*. Otherwise, we call it a *non-flaky build*. Figure 5.2 shows a box plot side-by-side comparing the APFDc for flaky builds and non-flaky builds under seven major TCP techniques (four change-based IR techniques: Tf-idf, BM25, LSI, LDA; two change-unaware techniques QTF, HIS and one hybrid technique CCHIR). We make the following observations:

1. All change-based IR techniques (Tf-idf, BM25, LSI, LDA) perform better on non-flaky tests than on flaky tests. As flaky failures are usually introduced into the program before the last revision [19] while non-flaky failures are more likely to be regressions introduced by the last change, therefore, change-based techniques are better at prioritizing failures that are non-flaky.

2. The QTF technique and HIS technique have a better performance on flaky tests than on non-flaky tests. This result suggests that in our dataset flaky tests tend to run faster and are more likely to have failed in the past.

3. Our hybrid technique CCHIR performs the best among the seven techniques on both flaky tests and non-flaky tests. This result demonstrates the robustness and effectiveness of our hybrid technique as it is positively influenced by the change-based technique without suffering from the weaknesses of it.

CHAPTER 6: THREATS TO VALIDITY

6.1 THREATS TO INTERNAL VALIDITY

The most important threat to the internal validity of our thesis is that our work is heavily depending on the third-party libraries and platforms. For example, the previous state of a build is passed or failed is copied from build log generated by Travis and that needs to be exactly the state of build at the code version before change. Also, we utilized TravisTorrent to help us parse build logs and the accuracy of failed test list is directly depending on the implementation quality of that tool.

Another threat to the internal validity is that we made fake failure-fault mappings in our evaluation because we cannot retrieve the real mappings between the failures and faults of the program builds in our dataset. This requires huge human efforts to inspect all the real code and we cannot afford to do this. Although we used a research question to study the impact of it, it still does not cover every possible case.

6.2 THREATS TO EXTERNAL VALIDITY

In our dataset the test FQNs, test outcomes and test execution times are extracted from the maven build logs, so whether we could get a complete and accurate dataset is highly depending on the completeness and the parsability of log files. Sometimes the maven logs are messed up because of concurrent tests or some other issues, resulting in a potential test missing or wrong test-to-time mapping.

CHAPTER 7: RELATED WORK

Test-case prioritization (TCP) has been proposed to reduce the time and cost of regression testing [13]. The idea is to prioritize tests that have a higher likelihood of detecting bugs to run first. Most prior work have implemented techniques based on test coverage, prioritizing tests that cover more, and diversity, ordering tests such that similar tests in terms of coverage are ordered later [36, 37, 18, 38, 39]. However, recent work shows that state-of-the-art coverage-based techniques are not cost effective at running the tests that detect bugs earlier, because they tend to execute long-running tests first [34]. To address this problem, one solution is to make new cost-cognizant coverage based techniques that are aware of both the coverage and cost of tests [33, 34]. In our work, we propose new hybrid TCP techniques that combine IR-based TCP techniques with test time and historical failure frequency. We find that these hybrid techniques perform better, leading to higher APFDc values.

There has been prior work that utilized information retrieval techniques in software testing [44, 45, 4]. Our work is most similar to prior work by Saha *et al.*, who proposed to prioritize tests using information retrieval techniques, ordering tests based on the similarities between tests and the program changes between two program versions [4]. Using IR to perform TCP can be effective because it does not require costly static or dynamic analysis but just needs light-weight textual-based computations. Saha *et al.* found that their IR-based TCP technique outperforms the state-of-the-art coverage-based techniques. However, Saha *et al.* evaluated their technique on a dataset with only eight projects and 24 version pairs, which potentially makes their result noninclusive and less generalizable. Also, they evaluate using APFD, which does not consider the cost of testing and does not reflect the effectiveness of TCP as well as APFDc. Our work improves upon Saha *et al.*'s evaluation by introducing a much larger dataset with real failures and real times, and we utilize APFDc to evaluate the effectiveness of IR-based TCP techniques.

Recently, several other works have used Travis CI in order to construct a large and real datasets to evaluate testing techniques [46, 31, 47, 48]. We also collect our dataset from Travis CI, collecting information from builds that involve real changes from developers and real test failures. With real failures used in evaluation, our work improves upon evaluation in other work that relies on mutation testing [41, 34, 49].

There has been much work in the area of flaky tests [19, 31, 50, 51]. In our work, we evaluate how TCP techniques perform on flaky tests and on non-flaky tests. From our evaluation, we find that the change-aware IR-based techniques perform better when considering only non-flaky test failures. However, we find that our hybrid TCP techniques are quite robust in the fact of flaky tests.

CHAPTER 8: CONCLUSIONS

We perform an extensive study to evaluate and enhance the recent program change-based, information-retrieval approach for test-case prioritization on a large-scale dataset with real test costs and real test failures. The major conclusions of our work include:

1. When evaluating TCP algorithms, it is important to evaluate them on a large-scale dataset, because the performance of techniques can differ a lot across different projects and even across different program builds in the same project.

2. IR techniques show a seemingly better performance when evaluated by cost-unaware metric APFD than by cost-aware metric APFDc, because the IR techniques we evaluated are designed to optimize the cost-unaware ranking of data objects ignoring the time costs of these objects.

3. Different TCP techniques compare differently when evaluated by different metrics. APFD is a not a very good measurement for real-world TCP whose objective is to save the time and cost for the developers, and an approach with a higher APFD does not necessarily offer more time saving.

4. Hybrid techniques, including cost-cognizant IR-based technique, and three factor hybrid technique, significantly outperforms the original IR technique when evaluated by APFDc. That is, the prior test execution times and historical failure frequency could enhance the original IR-based TCP.

5. Different TCP techniques compare the same under different failure-to-Fault mappings. That is, failure-to-Fault mapping is not a dominant variable when comparing TCP techniques.

6. Change-based TCP techniques perform better on non-flaky tests than on flaky tests, while hybrid techniques perform good on both non-flaky tests and on flaky tests. That is, change-based TCP is not good at detecting failures that are not caused by the program changes, while our hybrid techniques do not have the same weakness.

REFERENCES

[1] H. K. Leung and L. White, "Insights into regression testing (software testing)," in *Proceedings. Conference on Software Maintenance-1989*. IEEE, 1989, pp. 60–69.

[2] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 2017, pp. 233–242.

[3] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[4] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 268–279.

[5] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[6] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on software engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[7] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[8] O. Legunsen, A. Shi, and D. Marinov, "STARTS: STAtic Regression Test Selection," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 949–954.

[9] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2. IEEE, 2015, pp. 713–716.

[10] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 35–42, 2006.

[11] D. Jeffrey and N. Gupta, "Test suite reduction with selective redundancy," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 549–558.

[12] H.-Y. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *Proceedings of the 31st International Conference on Software Engineering.* IEEE Computer Society, 2009, pp. 419–429.

[13] 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.

[14] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360).* IEEE, 1999, pp. 179–188.

[15] 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.

[16] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta, "Assessing test case prioritization on real faults and mutants," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2018, pp. 240–251.

[17] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* ACM, 2014, pp. 437–440.

[18] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2016, pp. 559–570.

[19] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 643–653.

[20] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[21] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[23] S. E. Robertson, S. Walker, and M. Beaulieu, "Experimentation as a way of life: Okapi at TREC," *Information processing & management*, vol. 36, no. 1, pp. 95–108, 2000.

[24] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: BM25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[25] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* Valletta, Malta: ELRA, May 2010, http://is.muni.cz/publication/884893/en. pp. 45–50.

[26] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[27] X. Wei and W. B. Croft, "LDA-based document models for ad-hoc retrieval," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval.* ACM, 2006, pp. 178–185.

[28] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2016, pp. 426–437.

[29] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 2017, pp. 197–207.

[30] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[31] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering.* ACM, 2018, pp. 433–444.

[32] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering.* IEEE Computer Society, 2001, pp. 329–338.

[33] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," Technical Report TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln, Tech. Rep., 2006.

[34] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 2018, pp. 656–667.

[35] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* ACM, 2015, pp. 234–245.

[36] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[37] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 535–546.

[38] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 192–201.

[39] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 233–244.

[40] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive Test Selection," *arXiv preprint arXiv:1810.05286*, 2018.

[41] M. Azizi and H. Do, "ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 144–154.

[42] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 12–22.

[43] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 58–68.

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

[45] J.-H. Kwon, I.-Y. Ko, G. Rothermel, and M. Staats, "Test case prioritization based on information retrieval concepts," in *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, vol. 1. IEEE, 2014, pp. 19–26.

[46] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating test-suite reduction in real software evolution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 84–94.

[47] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: a study of Java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 821–830.

[48] M. Hilton, J. Bell, and D. Marinov, “A large-scale study of test coverage evolution,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 2018, pp. 53–63.

[49] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 654–665.

[50] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, “iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests,” in *Proceedings of the 12th IEEE International Conference on Sofware Testing, Verification, and Validation.* IEEE, 2019, pp. to–appear.

[51] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, “Detecting assumptions on deterministic implementations of non-deterministic specifications,” in *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016.* IEEE, 2016, pp. 80–90.