GTXCrawler: An Efficient System for Automated Test Case Selection in Continuous Integration Development Environment

Maral Azizi[a,*], Hyunsook Do[a]

[a] *University of North Texas, Denton, Texas, United States*

Keywords: Regression Testing, Test Case Prioritization, Software Repository, Test Quality Metrics.

1. Introduction

Modern software development tends to grow much faster than before, which has introduced the Continuous Integration (CI) concept. In CI practices, software systems involve frequently in a much faster pace comparing to the traditional software development techniques. Methodologies such as agile emphasize the importance of fast software deployment [9] and companies providing web based applications have taken this to extremes. For example, Amazon deploys software every 11.6 seconds on average [19]. Considering this fast pace of the software deployment makes the automatic regression testing an essential step.

Common regression testing techniques rely on static or dynamic code analysis. These techniques often apply multiple sources of information (e.g. code coverage, fault history, etc) to select the most important test cases that have high chance to detect faults. Although, many empirical studies have shown the effectiveness of these techniques [12, 7, 23, 16, 37, 36, 35, 17, 25, 13, 38], this information is usually unavailable before test cases are executed.

In code coverage based techniques, practitioners often select a test case based on the percentage of block or statement coverage. The underlying hypothesis of these technique is that the higher the code coverage of a test case is the more chance of catching a faults. However, we argue that this is not an optimum solution because: first, a test case with high code coverage might not cover the changed portion of the code, therefore, it might not be effective in catching faults. Second, collecting coverage information is a continuous task, which has to be repeated for each new version of a software and it can be very time consuming, particularly for large scale programs [24, 30]. Also, previously collected information is not accurate for a new program version due to the changes in source code and test suite.

Newly added test cases are another type of limitations of the traditional regression techniques. In traditional regression testing techniques, testers often select test cases based on their history and if they have failed history, they would be favored over other test cases. However, because there is no history information available for the new test cases, these tests would be ignored during the selection process regardless of their potential benefit for fault detection.

These limitations can be more problematic in CI environment where the speed of software delivery is one of the important success factors. Therefore, we argue that the traditional regression techniques will not be practical in modern software development practices due to following reasons: 1) the limited time for software deployment will not allow the code coverage or history data collection, 2) newly added test cases would be ignored despite their potential effectiveness, 3) the focus of testing might change over the time, therefore, some test cases might become more important in a certain period of time, and 4) the fault history information might not be useful for the new version of the system due to several changes, and added/deleted test cases. Moreover, instrumenting the system for code coverage monitoring requires significant amount of time and human knowledge. Therefore, the efficient testing approaches that can be aligned with today's fast release software practices should be considered.

[*] Corresponding authors
Email addresses: `maraazizi@my.unt.edu` (Maral Azizi), `hyunsook.do@unt.edu` (Hyunsook Do)

To address these limitations, in this paper, we propose an automated lightweight regression test case prioritization approach that utilizes information retrieval techniques that we name it GTXCrawler. Information retrieval (IR) techniques such as traceability link recovery techniques have been applied to solve various software engineering problems [31]. Common techniques for recovering traceability links among software artifacts analyze textual similarities among these artifacts. These techniques are often light-weight and efficient [**?**].

GTXCrawler, first parse the source files and test files then it converts them into a eXtensible Markup Language (XML) document type. It then generates traceability links between the source code and test cases in XML format. GTXCrawler then generates a graph of source code and test cases in which the nodes of the graph are classes and test cases and the edges represent the relations between them. Finally, GTXCrawler search algorithm selects test cases that exercise the most recent changes in the source code based on the graph dependency coverage.

To evaluate GTXCrawler, we used four open source software projects written in two different programming languages and compared it against three widely used test prioritization techniques. The experimental results show that our approach outperformed in majority of the application under test. The results show that GTXCrawler can be effective in practice by reducing a significant amount of time compared to the common techniques such as code coverage-based techniques. Thus, GTXCrawler provides an effective alternative approach to addressing the prioritization problem without requiring any dynamic coverage or static analysis. Furthermore, unlike traditional techniques, GTXCrawler is program language independent and can be applied to various programs written in different languages. The main contributions of our research are as follows:

- We introduce a new cost effective regression test case prioritization technique. Our proposed technique is language independent, fast, scalable, and light-weight, which generates a graph of source code and test cases and eliminates the cost of coverage profiling.
- Our proposed approach provides an efficient search algorithm to find the relevant test cases that exercise the change portion of the code through the generated graph.
- Our study also empirically evaluates the proposed approach by comparing it with three test prioritization techniques, which are commonly used for regression testing.

The rest of the paper is structured as follows. Section 2 explains the ReTEST approach. Section 3 outlines the research questions and the details of the experimental design. Section 4 presents the results of our study, and Section 6 discusses the threats to validity. Section 5 discusses the results including practical implications and guidelines for testers. Section 7 presents related work, and Section 8 discusses conclusions and future work.

2. Approach

Figure 1 represents the overview of tractability process, which includes four major steps as follow:

1. the grammatical tagging of the textual information of source and test files.
2. the conversion of tagged files to extensible markup language (XML).
3. the generation of tractability links among src-to-src, src-to-tst, and tst-to-tst files.
4. search the source files to identify the corresponding modules and test to portion a code changes.
5. results filtering and evaluation.

Below we explain each step in details.

2.1. Grammatical Tagging

The source and test files are usually written in structured forms of their programming languages. For instance, a class initially consist of libraries, variables, parameter, methods, etc. However, these parts of source files might vary depending on the grammatical rules of the programming languages.

Because our initial concern is to build a language independent system, we need to find a way to interpret each specific term into a common tag that will transfer the same meaning for each language. Therefor, our tagging process contains following steps: The first step, the parser read the document files and identifies the file type, which can be a

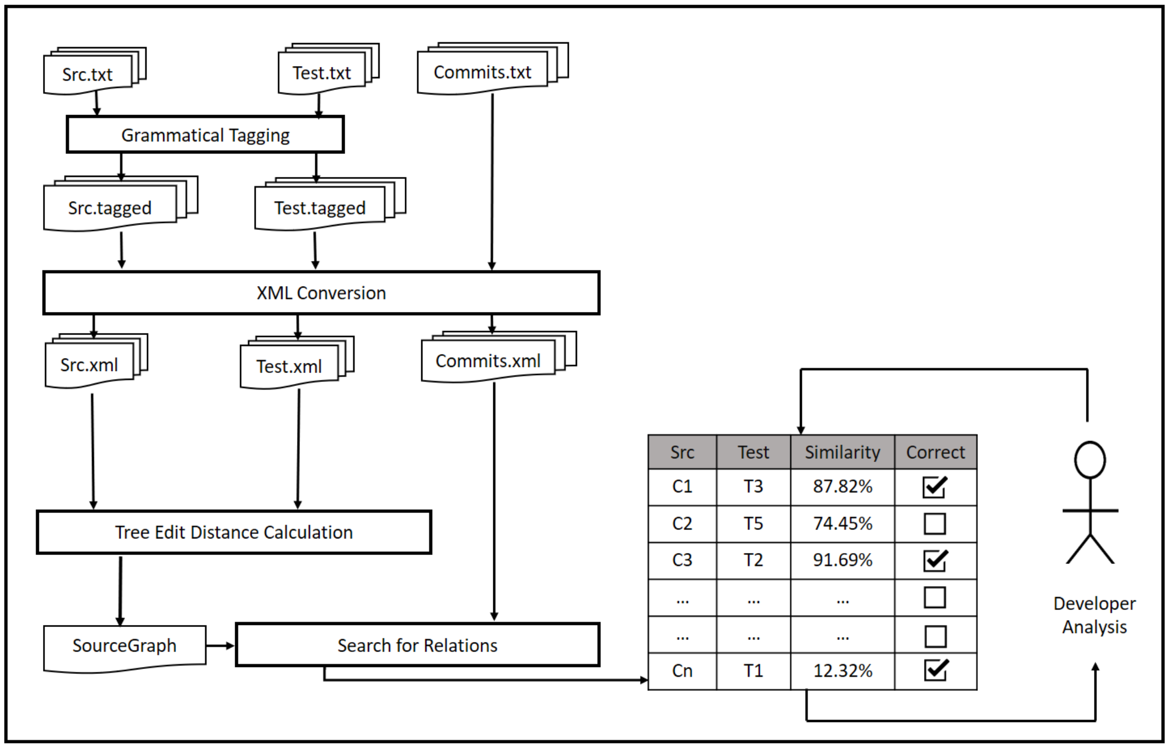


Figure 1: The Traceability Process

class, a library, a test class, etc. In the next step, initial tags are assigned to the words based on a lexicon and tagging rules. Then, the initial tags may be revised to take into account the context of words based on rules. For example, if a $< name >$ tag is followed by parenthesis in a source file, this tag name will be updated to a $< function >$ tag (see fig 2).

2.2. *XML Conversion*

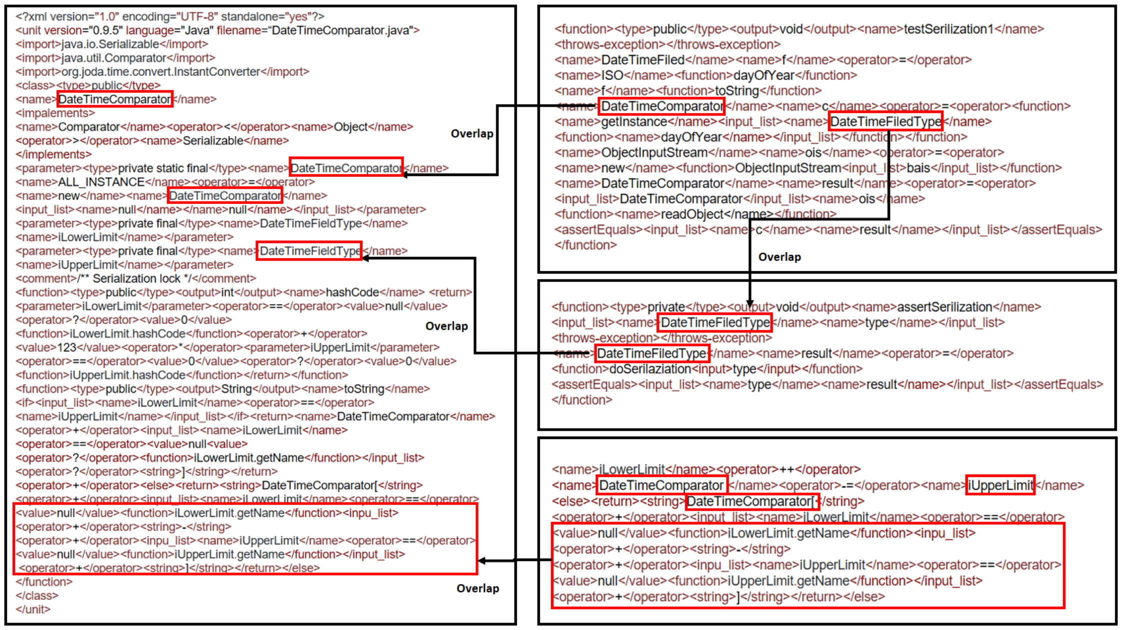
Once we built the tagged source code documents, we then transform each file into an equivalent XML document. Each XML document is structured according to the file type and contains elements that are used to illustrate the structure of the source files (e.g. elements that identify the abstract syntax tree (AST), in a source file that specify the parameters, methods, libraries, etc). The structure of source files and code commit would be similar because code commits are the modified versions of source files. Test files, however, include additional tags, which are related to the unit testing such as asserts, target class/method, etc.

Figure 2, shows the XML representation of the source and test methods of Figure 3. This representation reflects the structure of the original code structure. For instance, the description of the imported libraries of the class is enclosed in the XML element $< import >< /import >$. It represent the tags that have been assigned to individual key terms for each programming language (e.g. static, public, return, etc). Test files are converted to XML format in a similar way to the source files.

2.3. *Tractability Link Generation*

The next step is generating traceability links among the XML files. These set of relations are generated based on the analysis of the content of XML files, which are defined based on specific rules that we defined. To build these relations, we have specified the relations in three different categories as follows:

1. src-to-src dependency: These set of dependencies are designed to determine the relations among source files, which specify the way of matching syntactically related terms in the textual parts of a source file with semantically related elements of the other source files.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit version="0.9.5" language="Java" filename="DateTimeComparator.java">
<import>java.io.Serializable</import>
<import>java.util.Comparator</import>
<import>org.joda.time.convert.InstantConverter</import>
<class><type>public</type>
<name>DateTimeComparator</name>
<impalements>
<name>Comparator</name><operator><</operator><name>Object</name>
<operator>></operator><name>Serializable</name>
</implements>
<parameter><type>private static final</type><name>DateTimeComparator</name>
<name>ALL_INSTANCE</name><operator>=</operator>
<name>new</name><name>DateTimeComparator</name>
<input_list><name>null</name></name>null</name></input_list></parameter>
<parameter><type>private final</type><name>DateTimeFieldType</name>
<name>iLowerLimit</name></parameter>
<parameter><type>private final</type><name>DateTimeFieldType</name>
<name>iUpperLimit</name></parameter>
<comment>/** Serialization lock */</comment>
<function><type>public</type><output>int</output><name>hashCode</name> <return>
<parameter>iLowerLimit</parameter><operator>==</operator><value>null</value>
<operator>?</operator><value>0</value>
<function>iLowerLimit.hashCode</function><operator>+</operator>
<value>123</value><operator>*</operator><parameter>iUpperLimit</parameter>
<operator>==</operator><value>0</value><operator>?</operator><value>0</value>
<function>iUpperLimit.hashCode</function></return></function>
<function><type>public</type><output>String</output><name>toString</name>
<if><input_list><name>iLowerLimit</name><operator>==</operator>
<name>iUpperLimit</name></input_list></if><return><name>DateTimeComparator</name>
<operator>+</operator><input_list><name>iLowerLimit</name>
<operator>==</operator><value>null<value>
<operator>?</operator><function>iLowerLimit.getName</function></input_list>
<operator>?</operator><string>]</string></return>
<operator>+</operator><else><return><string>DateTimeComparator[</string>
<operator>+</operator><input_list><name>iLowerLimit</name><operator>==</operator>
<value>null</value><function>iLowerLimit.getName</function><inpu_list>
<operator>+</operator><string>-</string>
<operator>+</operator><inpu_list><name>iUpperLimit</name><operator>==</operator>
<value>null</value><function>iUpperLimit.getName</function></input_list>
<operator>+</operator><string>]</string></return></else>
</function>
</class>
</unit>
```

```
<function><type>public</type><output>void</output><name>testSerilization1</name>
<throws-exception></throws-exception>
<name>DateTimeFiled</name><name>f</name><operator>=</operator>
<name>ISO</name><function>dayOfYear</function>
<name>f</name><function>toString</function>
<name>DateTimeComparator</name><name>c</name><operator>=<operator><function>
<name>getInstance</name><input_list><name>DateTimeFiledType</name>
<function><name>dayOfYear</name></input_list></function></function>
<name>ObjectInputStream</name><name>ois</name><operator>=<operator>
<name>new</name><function>ObjectInputStream<input_list>bais</input_list></function>
<name>DateTimeComparator</name><name>result</name><operator>=<operator>
<input_list>DateTimeComparator</input_list><name>ois</name>
<function><name>readObject</name></function>
<assertEquals><input_list><name>c</name><name>result</name></input_list></assertEquals>
</function>
```

```
<function><type>private</type><output>void</output><name>assertSerilization</name>
<input_list><name>DateTimeFiledType</name><name>type</name></input_list>
<throws-exception></throws-exception>
<name>DateTimeFiledType</name><name>result</name><operator>=</operator>
<function>doSerilaziation<input>type</input></function>
<assertEquals><input_list><name>type</name><name>result</name></input_list></assertEquals>
</function>
```

```
<name>iLowerLimit</name><operator>++</operator>
<name>DateTimeComparator </name><operator>-=</operator><name>iUpperLimit</name>
<else><return><string>DateTimeComparator[</string>
<operator>+</operator><input_list><name>iLowerLimit</name><operator>==</operator>
<value>null</value><function>iLowerLimit.getName</function><inpu_list>
<operator>+</operator><string>-</string>
<operator>+</operator><inpu_list><name>iUpperLimit</name><operator>==</operator>
<value>null</value><function>iUpperLimit.getName</function></input_list>
<operator>+</operator><string>]</string></return></else>
```

Figure 2: XML representation of the source code and test cases of figure 3 with their traceability relations

Due to the different of grammatical and tagging structure of each programming language, matching at this stage may also be informed by synonym lexicons. For example, $\#include < x >$ in c language is equivalent to $import\ x$ in Java. The synonym lexicon file will interpret all related terms to a single unique tag (e.g. $< import >< /import >$ represent library import regardless of their initial programming language).

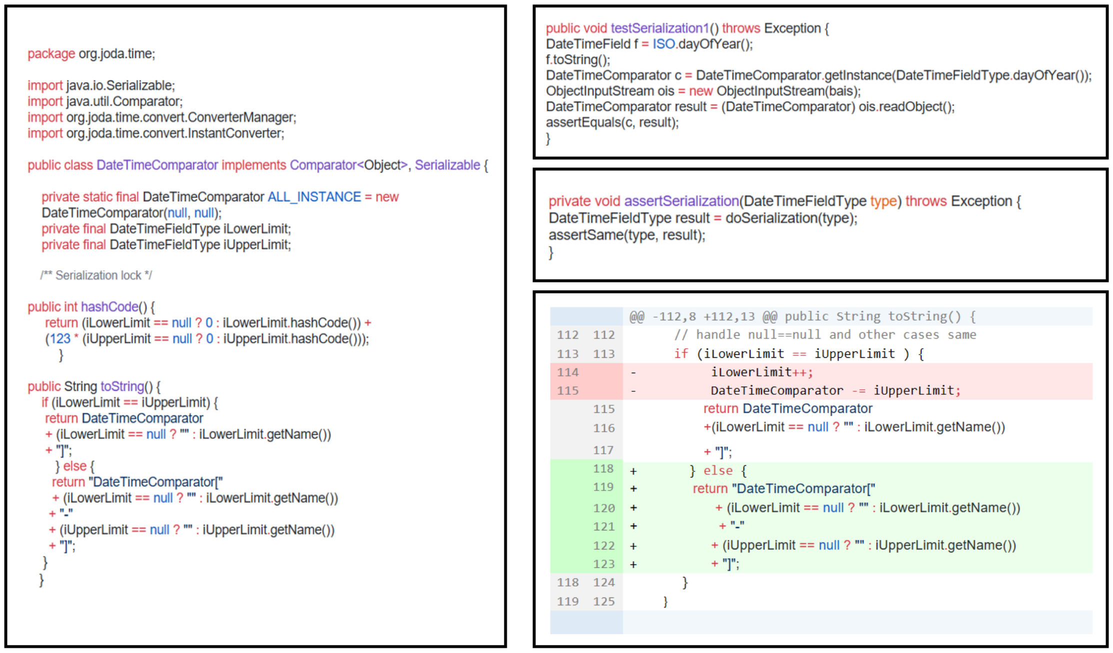
2. tst-to-src dependency: Relational links between test and source files are designed similar to the tst-to-tst link. However, the linkage are between source and test files. In order to identify whether a test case can be related to a source file we use the same notion, which is whether the same string parameter has appeared in both files. The underlying hypothesis is that we assume that specific term could be a function or parameter name and if it appears in both test and source file it means that this test exercise that portion of the source code.
3. tst-to-tst dependency: The other type of traceability relations are among test cases, which can be the relations between separate test methods of a single or different test classes. Similarly to other dependencies, these dependency links are specified in XML ($test - links.xml$). The condition that may be used to specify the tst-to-tst is whether two string parameters are equal in two different test cases. Therefore, we can generate a relational link between these two test cases. Figure 2 shows an example of how test cases can be related to each other.

All above mentioned types of dependencies are generated using Tree Edit Distance (details are explained in 2.3.1).

2.3.1. Source Graph Generation and Search Algorithm

First we need to generate the relational links among source to test cases, we then build a graph in which the nodes are source files and test cases and the edges represent the relation among the nodes. Figure 4 shows an example of generated source graph. Because the initial goal of our approach is to automate the regression test case prioritization, we need to search among the source graph to determine the portion of the source code that correspond to the code changes. To do that, we use the code changes between two program versions that we have converted them to XML files (see Fig 3 and 2) as a input for the search in the source graph.

The tree edit distance algorithm is a widely used technique by researcher to compare semi-structured data, such as XML documents [33, 10]. Therefore, we apply this algorithm to calculate the similarity between code commits and each source file. This algorithm find the minimal cost sequence of edit operation that can transfer a labeled tree to another one. More formally, for a given labeled tree T where each node is a symbol from a fixed finite alphabet $\sum$. We call T an ordered tree if a left-to-right order among siblings in T is given. The matching problems is simple primitive operations applied to labeled trees. For an ordered tree T, these set of edit operations are:



```
package org.joda.time;

import java.io.Serializable;
import java.util.Comparator;
import org.joda.time.convert.ConverterManager;
import org.joda.time.convert.InstantConverter;

public class DateTimeComparator implements Comparator<Object>, Serializable {

   private static final DateTimeComparator ALL_INSTANCE = new
   DateTimeComparator(null, null);
   private final DateTimeFieldType iLowerLimit;
   private final DateTimeFieldType iUpperLimit;

   /** Serialization lock */

public int hashCode() {
    return (iLowerLimit == null ? 0 : iLowerLimit.hashCode()) +
    (123 * (iUpperLimit == null ? 0 : iUpperLimit.hashCode()));
      }

public String toString() {
   if (iLowerLimit == iUpperLimit) {
    return DateTimeComparator
    + (iLowerLimit == null ? "" : iLowerLimit.getName())
    + "]";
      } else {
     return "DateTimeComparator["
     + (iLowerLimit == null ? "" : iLowerLimit.getName())
    + "-"
    + (iUpperLimit == null ? "" : iUpperLimit.getName())
    + "]";
   }
  }
```

```
public void testSerialization1() throws Exception {
DateTimeField f = ISO.dayOfYear();
f.toString();
DateTimeComparator c = DateTimeComparator.getInstance(DateTimeFieldType.dayOfYear());
ObjectInputStream ois = new ObjectInputStream(bais);
DateTimeComparator result = (DateTimeComparator) ois.readObject();
assertEquals(c, result);
}
```

```
private void assertSerialization(DateTimeFieldType type) throws Exception {
DateTimeFieldType result = doSerialization(type);
assertSame(type, result);
}
```

```
@@ -112,8 +112,13 @@ public String toString() {
        // handle null==null and other cases same
        if (iLowerLimit == iUpperLimit ) {
-           iLowerLimit++;
-           DateTimeComparator -= iUpperLimit;
           return DateTimeComparator
           +(iLowerLimit == null ? "" : iLowerLimit.getName())

           + "]";
+       } else {
+        return "DateTimeComparator["
+             + (iLowerLimit == null ? "" : iLowerLimit.getName())
+              + "-"
+            + (iUpperLimit == null ? "" : iUpperLimit.getName())
+            + "]";
         }
      }
```

Figure 3: An example of class, two test case, and a code commit. testSerilization() is from TestDateTimeComparator.java class, and assertSeilization() is from TestDateTimeFieldType.java class.

- rename: Change the label of a vertex v in T.
- delete: Delete a non-root vertex v in T with parent v', making the children of v become the children of v'.
- insert: Insert a vertex v as a child of v' in T making v the parent of a consecutive subsequence of the children of v'.

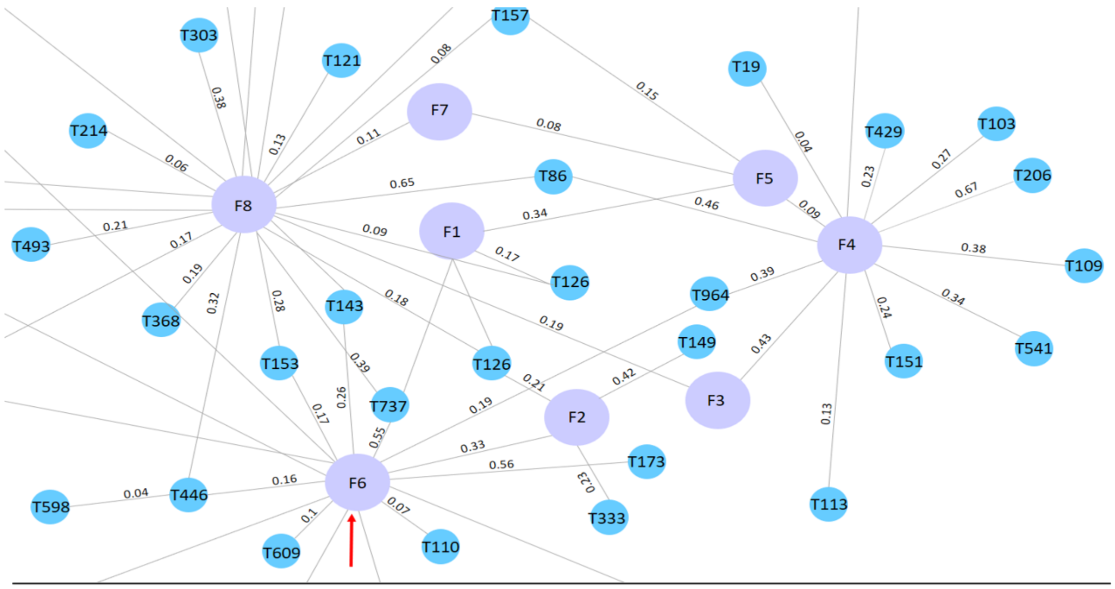
For a given cost function defined for each edit operation. An edit distance D between T1 and T2 is a sequence of edit operations turning T1 into T2. The cost of D is the sum of the costs of the operations in D. The tree edit distance problem is to calculate the edit distance and a corresponding edit script [10]. Once we identify the corresponding portion of the code, we then select all linked test cases that are directly or indirectly exercising that portion of the code change. Below we explain test selection process in detail.

Figure 4 represents an example of the generated source code graph. In this example, if function f6 has been modified (pointed with an array), therefor its change can impact function f2 too because f2 is dependent on f6. GTXCrawler starts with identifying all related test cases to the modified file and all other files that are dependent on the modified file (in this case F2). It then returns a report of the dependent test cases with their similarity scores. At this point developer reviews the test cases to evaluate a suitable similarity scores that can be a good indicator to the test files and filters out those that not related. Generally, the higher similarity scores indicates that there is a higher correlation between files.

Assume that developer set the minimum similarity score to 0.15, which means that only test cases that meet this minimum score should be selected. GTXCrwaler then reorder the test cases in this way: first it selects all dependent test cases to both F6 and F2 that can satisfy the minimum similarity score, it then prioritizes the test cases based on their similarity scores. The test cases that have higher similarity score will be selected earlier because we hypothesis that those test cases have higher correlation to the modified files. Therefore, the set of test cases to be selected first is T173, T737, T149, T143, T333, T126, T964, T153, AND T446.

3. Empirical Study

In this paper, we investigate whether the use of GTXCrawler can improve the effectiveness of regression test prioritization techniques. To assess our proposed technique, we performed an empirical study. The following subsections



Rank	1	2	3	4	5	6	7	8	9
Test Id	T173	T737	T149	T143	T333	T126	T964	T153	T446
Similarity Score	0.56	0.55	0.42	0.26	0.23	0.21	0.19	0.17	0.16

Figure 4: An Example of Source Graph Network Layout

present our objects of analysis, data collection, study setup, and threats to validity. In particular, we address the following research questions:

- RQ1: IS GTXCrawler effective in identifying relevant test cases to the changed files?
- RQ2: How effectively can GTXCrawler conduct priority based test selection, compared to control techniques?
- RQ3: Can GTXCrawler be useful for improving test case prioritization techniques?

3.1. Objects of Analysis

In this study, we used four open source programs. Table 1 lists the applications and their associated data: “Version pair” (the program versions that we used for the experiment), “LOC” (the number of lines of code), “TstMethod” (the number of test cases in method level), “TstClass” (the number of test cases in class level), “Faults” (the number of faults), “Queries” (the number of queries that we built from the program change files), “SrcToken” (the number of tokens of source code), and “TstToken” (the number tokens of test files).

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

[1]The reported faults for *nopCommerce* are available in the *GitHub* repository [4], and the bug history for *Umbraco* is accessible through their website [5].

Table 1: Subject Applications Properties

No	Object	Version pair	LOC	# TstMethod	# TstClass	# Faults	# Queries	# Elements	# Attributes
P_1	nopCommerce	2.00 - 2.10	133,862	523	126	28	65	17803	2768
P_2	nopCommerce	2.20 - 2.30	141,232	585	136	32	118	18052	2893
P_3	Umbraco-CMS	7.7.0 - 7.7.2	312,511	6,025	628	41	97	26700	14552
P_4	Umbraco-CMS	7.7.4 - 7.7.5	312,886	6,026	628	28	131	26811	14546
P_5	Joda-Time	0.9 - 0.95	17,818	923	80	10	143	3328	2767
P_6	Joda-Time	0.98 - 0.99	20,317	3,181	218	10	128	4905	3765
P_7	Joda-Time	1.1 - 1.2	22,149	3,360	237	21	130	5290	4368
P_8	Joda-Time	1.2 - 1.3	24,001	4,256	254	21	115	5380	4622
P_9	Joda-Time	1.3 - 1.4	25,292	4,511	263	12	54	5496	4812
P_{10}	Joda-Time	1.4 - 1.5	25,795	4,701	266	18	112	5610	5106
P_{11}	JFreeChart	1.0.0 - 1.0.1	95,669	1,512	225	12	98	14219	6039
P_{12}	JFreeChart	1.0.2 - 1.0.3	101,713	2,151	231	10	6	11855	4750
P_{13}	JFreeChart	1.0.4 - 1.0.5	74,434	2,360	351	13	101	12057	5102
P_{14}	JFreeChart	1.0.6 - 1.0.7	76,065	2,680	366	8	85	12662	5522
P_{15}	JFreeChart	1.0.8 - 1.0.9	81,541	2,738	393	11	186	12738	5636

3.2. Variables and Measures

3.2.1. Independent Variables

The independent variable in this study is regression test prioritization technique. We considered three test case prioritization techniques, which we classified them into two groups: control and heuristic. Below we summarized these groups and techniques. For our heuristic techniques, we used the approach explained in Section 2, so, here, we only explain the control techniques.

1. Total Statement (T): This technique prioritizes test cases based on the total number of statements they cover. If multiple tests cover the same number of statements, they are ordered randomly.
2. Additional Statement (A): This technique prioritizes test cases based on the number of additional statements they cover. If multiple test cases cover the same number of statements not yet covered, they are ordered randomly.
3. Test Dissimilarity (D): This technique prioritizes test cases using the dissimilarity among test cases. To measure the dissimilarity, we used the Jaccard Index.

3.2.2. Dependent Variable and Measures

RQ1 and RQ2 is percision and recall and f-measure

The dependent variable for RQ3 is average percentage of fault detection (APFD), which measures the average percentage of faults detected during the test suite execution. The range of APFD is from 0 to 100; higher values indicate faster fault detection rates. Given T as a test suite with n test cases and m number of faults, F as a collection of detected faults by T, and TF_i as the first test case that catches the fault i, we calculate APFD [29] as follows:

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

The primary concern of the RQ2 is to compare the code coverage of the GTXCrawler with that of other control techniques in term of time efficiency. Therefore, the dependent variable for RQ2 is the average percentage of code coverage without and with GTXCrawler.

3.3. Data Collection and Experimental Setup

As described earlier, GTXCrawler does not require any dynamic or static code coverage information, and it calculates the similarity among code commits and source code/test cases. To collect code coverage information for control techniques we used Visual Studio Test Analysis plugin for nopCommerce and Umbraco. For other two Java applications, we used JaCoCo plugin on netbeans IDE [6]. We also collected test execution time using a PC with CPU Core i7, memory 16 GB, and operating system Windows 10.

To compare the result of GTXCrawler with the code coverage techniques we built a greedy algorithm that selects those set of test cases that are exercising the modified portion of code and then we prioritize them based on their percentage of the code that they cover. For diversity based technique we also measure the Jaccard Index among those set of test cases that are exercising the changed portion of code, then we prioritized them based on their distance from each other.

4. Data and Analysis

4.1. RQ1 Analysis

In RQ1, we evaluate GTXCrawler ability to see if it can accurately detect the relationship between the classes to the test cases. To do this, we compared the results of applying GTXCrawler with Jaccard Index technique, which is a commonly used technique for calculating the term similarity in test suite [25]. We choose Jaccard Index because the previous study shown that this technique is a better term similarity indicator than other techniques [24].

To evaluate the effectiveness of GTXCrawler, we used common accuracy indicators to determine the accuracy of our model. The three accuracy indicators that we used are precision (Prec.), Recall and F-Measure (FM). Prec indicates the percentage of correctly classified instances, Recall indicates the percentage of correctly selected test cases that are relevant to the modified source file, and FM is the weighted average of Precision and Recall. We randomly select 20 percent of the test cases as a test dataset and 60 percent as a training dataset and 20% as validation set.

Table 2 shows the average results for each application. The results show that the Recall, Precision and F-measure achieved by GTXCrawler is statistically higher than that achieved by Jaccard Index in most cases. Only in four out of fifteen cases Jaccard Index achieved higher f-Measure (Umbraco-CMS v7.7.2, Joda-Time v1.2, Joda-Time v1.3 and Joda-Time v1.4). Therefore, we can conclude that GTXCrawler is a better tractability link indicator than Jaccard Index.

Table 2: Accuracy Results Achieved Using GTXCrawler Versus Jaccard Index.

Application	GTXCrawler			Jaccard Index		
	Prec. (%)	Recall(%)	FM (%)	Prec. (%)	Recall(%)	FM (%)
nopCommerce v2.10	88	76	81	69	52	59
nopCommerce v2.30	81	77	78	67	71	69
Umbraco-CMS v7.7.2	74	80	76	80	77	78
Umbraco-CMS v7.7.5	83	81	82	79	74	76
Joda-Time v0.95	77	72	74	57	59	58
Joda-Time v0.99	72	76	74	66	72	69
Joda-Time v1.2	69	63	66	73	78	75
Joda-Time v1.3	80	72	75	85	81	83
Joda-Time v1.4	81	78	80	84	80	82
Joda-Time v1.5	80	87	83	59	71	64
JFreeChart v1.0.1	92	92	92	74	65	69
JFreeChart v1.0.3	83	78	80	69	58	63
JFreeChart v1.0.5	83	81	82	70	64	67
JFreeChart v1.0.7	67	77	72	72	66	69
JFreeChart v1.0.9	79	81	80	63	69	66

4.2. RQ2 Analysis

Figure 5 shows how many tests need to be selected by each approach to achieve a given recall average. For instance, in case of nopCommerce V2.10 our approach requires 251 (0.48%) top-ranked tests to detect 100% of the test failures, compared to at least 407 (78%) tests when using the total code coverage technique. In case of JfreeChart V1.0.7 our approach requires 1,206 (45%) top-ranked tests to detect 75% of the test failures, compared to at least 1,608 (60%) tests when using the Jaccard Index technique.

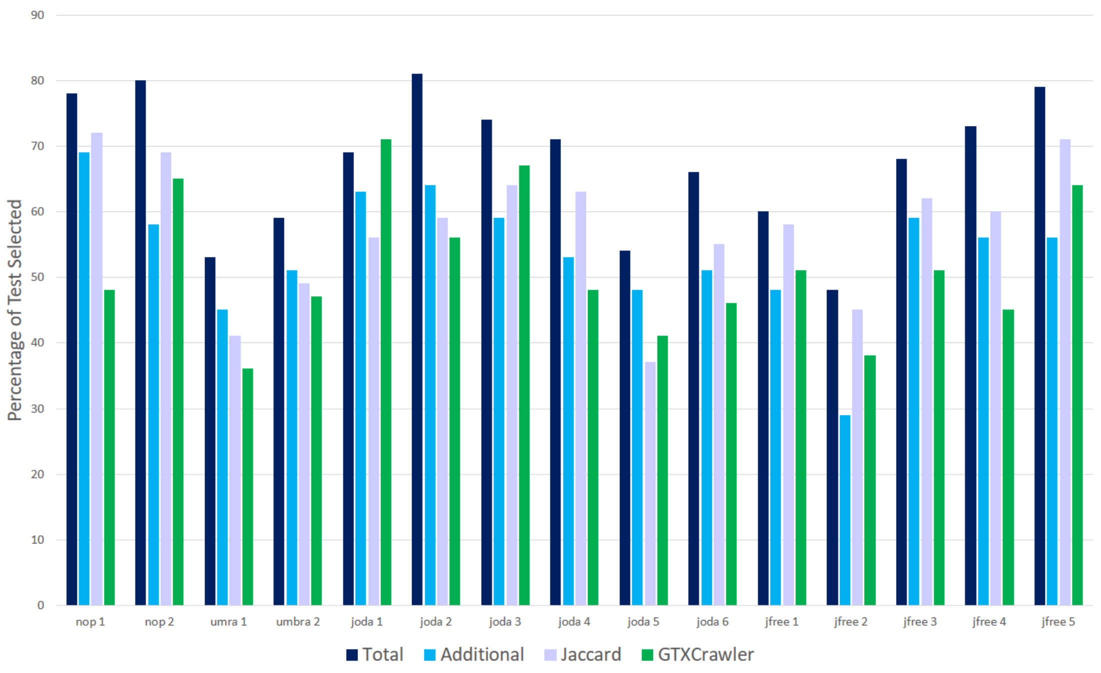
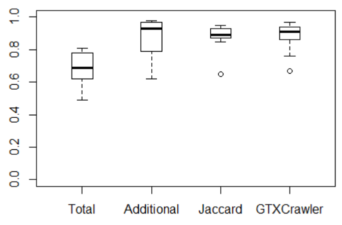


Figure 5: Average percentage of test selected

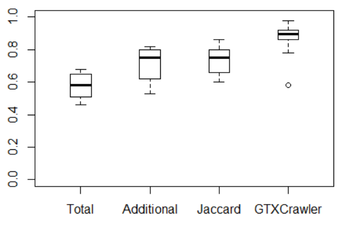
4.3. RQ3 Analysis

RQ3 investigates the applicability of GTXCrawler in the area of test case prioritization. To evaluate the effectiveness of GTXCrawler we computed the APFD values and compared its results with three different prioritization techniques. Figure 6 shows the APFD results variation. The horizontal bar shows the prioritization technique and the vertical bar represent APFD values.

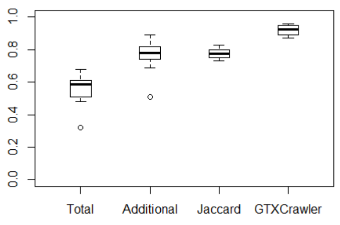
This experiment was performed 30 times with random selections each time. APFD values were computed by prioritizing all test cases for each application. In eleven of the fifteen subject application, prioritization by GTX-Crawler yielded a higher APFD than other prioritization techniques. However, all the APFD values were within a few percentage point of each other.



nopCommerce V2.10



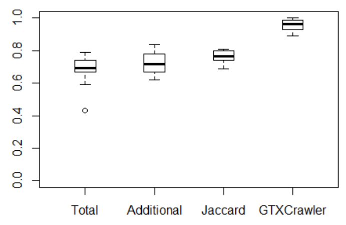
nopCommerce V2.30



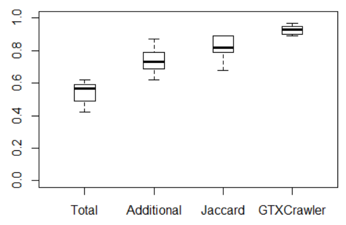
Umbraco V7.7.2

5. Discussion

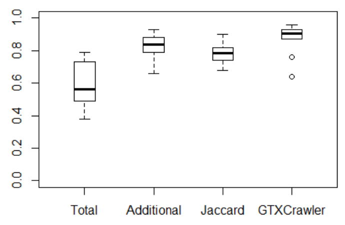
(* HD: Lets' cut the discussion of code coverage-based approach problems because the intro talks about them already. Instead, we need to focus on discussing the results in depth and their implications. As a journal, the current discussion is really short and it's somewhat very similar to issre18 paper. Please think about how to extend it by looking at the journal we wrote with Jeff or my other journals. *)



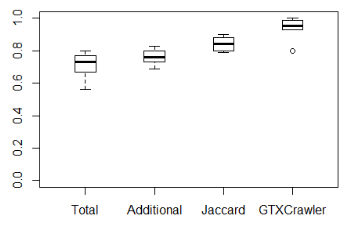
Umbraco V7.7.5



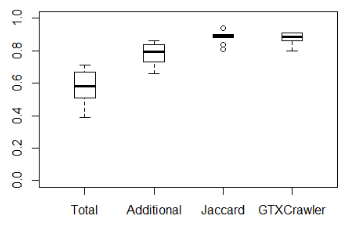
Joda-Time V 0.95



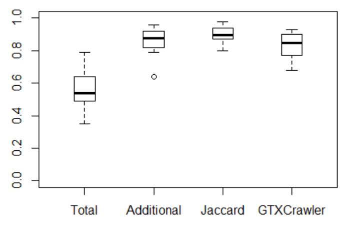
Joda-Time V 0.99



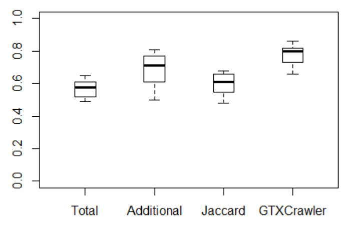
Joda-Time V 1.2



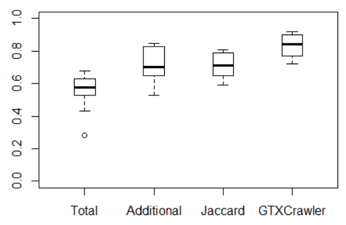
Joda-Time V 1.3



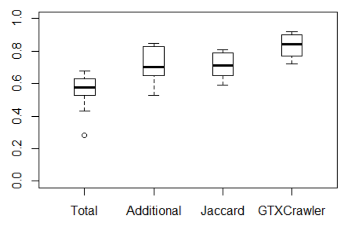
Joda-Time V 1.4



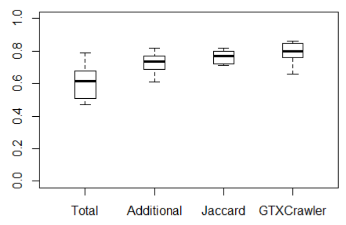
Joda-Time V 1.5



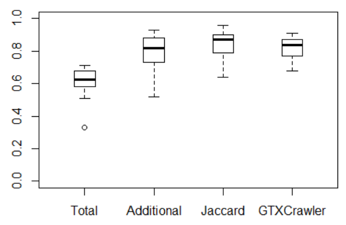
JfreeChart V1.0.1



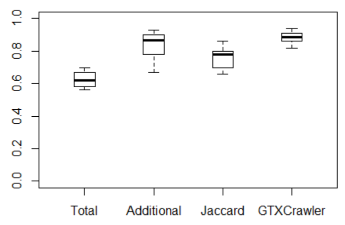
JfreeChart V1.0.3



Joda-Time V 1.0.5



Joda-Time V 1.0.7



JfreeChart V1.0.9

Figure 6: Mean percentage of faults detected over time, where time is shown in percentage of total execution time of all the test cases, in the test suite. The error bars correspond to the 95% confidence interval.

The results of our empirical study show that usage of GTXCrawler improves the regression testing. Code coverage-based techniques are one of the most widely used regression testing techniques. In this technique, the selection/prioritization algorithm is often based on the percentage of code covered by a test case. This technique hypothesizes that the test cases that are covering a higher portion of the code have the higher possibility of finding bugs. However, these techniques often have significant overhead and has to be repeated for very new program version. These techniques are more problematic in large scale programs with thousands of lines of code and test cases. Therefore, this technique can not be aligned with modern software development practices where the rate of software release is very fast such as agile system development and continuous integration systems []. However, shifting from traditional regression testing techniques to IR-based techniques helps to reduce the cost overhead by eliminating the coverage profiling. In this research, we have shown that the application of IR-based technique to regression testing techniques is advantageous, practical with continuous integration system development practices, and we have demonstrated concrete examples of how it can be applied.

One major benefit of applying GTXCrawler is that it's performance remains nearly constant even with the growth of the dataset. This can be more advantageous for modern software development, where the program undergoes many changes and the number of test cases grow rapidly. For instance, in case of Joda-Time the number of test cases increase by 409% comparing version 0.95 to version 1.5. Also, in cases when the size of a program increases, it does not effect the performance of GTXCrawler because each query is localized to a portion of the graph database. For instance, it took 49 seconds to select tests from the graph with 1,512 test cases of JFreeChart V1.0.0 using 98 queries, it only took 73 seconds to select a set of test cases from 2,738 test cases (1.8 times larger) of JFreeChart V1.0.9 for 186 queries.

Moreover, our results indicate that the differences between heuristic and control techniques are statistically significant. To see whether the differences we observed through the statistical analyses are practically meaningful we measured the effect sizes of differences. Table 3 presents the effect sizes of differences between control and heuristic techniques. As shown in Table 3, the effect sizes range from 0.44 to 4.18, which are considered to be small (JfreeChart: GTXCrawler in comparison with Jaccard Index) and large (Umbraco: GTXCrawler in comparison with Total) effect sizes, so we can say that the differences we observed in our studies are indeed practically significant.

Table 3: Experiment Objects and Associated Data.

Application	GTXCrawler vs Total	GTXCrawler vs Additional	GTXCrawler vs Jaccard Index
nopCommerce	3.97	1.56	1.48
Umbraco-CMS	4.18	3.16	3.50
Joda-Time	2.52	1.06	0.59
JFreeChart	2.47	0.49	0.44

6. Threats to Validity

The main threat to validity is the technique we used to calculate the term similarity. As we explained in the approach section we applied IR technique to calculate the similarity of code changes and test cases. The underlying hypothesis of this technique is that testers use similar terms in test as they use in a program source code. Therefore, this technique is not applicable for different types of testing such as GUI test etc.

One threat to the validity in this study is the choice of the test diversity measure. Although in our approach we used a commonly applied measure for XML documents (Tree Edit Distance), the results can be affected based on the selected algorithm. This limitation can be addressed through additional empirical studies with different algorithms.

Also, the choice of the similarity value can affect the results. In this experiment we set the average similarity score obtained for each application per query. Similarity scores for most applications varies from 0.03 to 0.62. However, our suggested algorithm is flexible and these values could be adjusted depending the amount of time, available resources, program characteristics (e.g, number of lines of code changed, number of test cases, etc) or organization preferences.

Another threat to validity is the generalization of our results. In this study, we used four open-source applications, so the findings from our results cannot be interpreted in the context of industrial applications. However, we tried to reduce this threat by using relatively large applications (17K to 312K LOCs) with large numbers of test cases (208 to 6,026) with real faults. Nevertheless, this threat can be further addressed by utilizing industrial software systems from various application domains.

7. Related Work

7.1. *Test case prioritization for regression testing*

For more than two decades, reducing the time and cost of regression testing has been an active research topic. Recent surveys on regression testing techniques [11, 36] provide a comprehensive understanding of overall trends of the techniques and areas for improvement. To date, numerous regression testing techniques have been implemented using various types of data sources (e.g., code coverage, test case diversity, and fault history), the code coverage-based technique is one of the most widely used and evaluated techniques. Although the code coverage-based approach is naïve and easy to implement, many empirical studies have shown that it can be effective [14, 15, 16, 26]. For instance, Leon and Podgurski [20] compared coverage-based selection with distribution of test execution profiles. The results of their empirical analysis show that the distribution based techniques can be as efficient or more efficient for revealing defects than coverage-based techniques, but that the two kinds of techniques are also complementary in the sense that they find different defects. Despite the effectiveness of aforementioned approaches, these techniques suffer from the computational overhead and the demand for collecting and analyzing different test quality metrics (e.g. code coverage, fault history, etc), which makes them less practical for continuous integration software environment.

Recently some researchers have invetigated modern techniques that can be aligned with CI environment. For example, Spieker et al [32] proposed an automatic learning test prioritization method in CI, which uses failure history information as input of the reinforcement learning system to select and prioritize test cases. In another work by Marijan et al [22], they presented a case study of a test prioritization approach for to improve the efficiency of continuous regression testing of industrial video conferencing software. their proposed approach prioritizes test cases using a weighted function, which calculates test weigh based on historical failure data, test execution time and domain-specific heuristics. By contrast, SANI does not require any type of historical or code coverage information and it is a lightweight, scalable, and language independent method, which only uses the source code and test files and generated the links between different portions of the code.

7.2. *Tractability Link Recovery*

Tractability link recovery have been investigated heavily by researchers in different areas of software engineering such as requirements, maintenance, etc. Most of these approaches are heuristic based or IR based [27]. In [31] authors propose an rule-based engine approach for requirement document traceability rule generation. Similar approaches have been proposed for automated update of traceability relations between requirements, analysis and design models of software systems [21, 8]. In [28], Rodriguez and Carver compared the performance of three major traceability techniques: information retrieval, Vector Space Model, and Latent Semantic Indexing for requirement traceability recovery. Their results show that Latent Semantic Indexing produces lower precision and recall compared to probabilistic IR and Vector Space Model.

In another work by Qusef et al. [27], authors proposed a traceability link recovery approach that exploits dynamic slicing to identify a set of candidate tested classes to Junit tests. The applied dynamic slicing method to identify the class of the reference variables. They have only focused on the last assert statement in Junit test, while GTXCrawler parses the entire test script and source code to identify all possible relationship between tests and code. Therefore, we argue that GTXCrawler would be more accurate to identify the relations.

Further, some researcher propose applying data mining techniques for traceability link recovery between source code artifacts. For instance, Zimmermann et al. applied association rule mining technique to help programmers identify changes within source code automatically [39]. Ying and Murphy proposed data mining approach, which extract patterns of changes in a software repository to help developers identify pertinent source. They have evaluated their techniques using Eclipse and Mozilla as subject programs. The results of this experiment show that the proposed technique can reveal valuable dependencies that may not be obvious from other existing analyses [34].

8. Conclusions and Future Work

To date, researcher and practitioners have proposed various techniques to reduce the cost of regression testing. However, existing techniques mainly rely on either dynamic coverage information or static program analysis, which they often have significant cost of overhead and are imprecise. In this research, we have introduced a novel technique, GTXCrawler, to improve the effectiveness as well as reducing the cost of regression testing. GTXCrawler uses the textual similarity of a program source code and returns a list of test cases that are are highly likely covering the modified portion of the program. GTXCrawler does not need any dynamic code coverage or static analysis and also it is flexible, scalable, and language independent. We evaluated our approach using four open-source software projects with three other test prioritization techniques. Our empirical results indicate that the use of the GTXCrawler can reduce the the cost of regression testing considerably.

Because our initial attempt to apply GTXCrawler in the area of regression testing has shown promising results, we aim to investigate this approach further by considering various algorithms (e.g., a hybrid ranking algorithm), various characteristics of applications (e.g., fault history information), different application domains (e.g., mobile applications), and various testing contexts (e.g., different testing processes or environment). For future work, we wish to continue expanding on this research by adding a learner algorithm to the framework that can help the system to be smarter and more efficient by learning from it's success and failures. Moreover, we wish to expand on this research as more data becomes available across a wider range of applications and different metrics. We also want to investigate how this research applies to different areas of regression testing such as test case selection and reduction.

References

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

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

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

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

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

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

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

[8] Gabriele Bavota, Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. Enhancing software artefact traceability recovery processes with link count information. *Information and Software Technology*, August 2013.

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

[10] P. Bille. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science*, June 2005.

[11] S. Biswas, R. Mall, M. Satpathy, , and S. Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011.

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

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

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

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

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

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

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

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

[20] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *in Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, pages 442–456. IEEE, 2003.

[21] Patrick Mader and Orlena Gotel. Towards automated traceability maintenance. *The Journal of Systems and Software*, October 2011.

[22] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test case prioritization for continuous regression testing an industrial case study. In *International Conference on Software Maintenance*, 2013.

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

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

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

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

[27] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *The Journal of Systems and Software*, September 2013.

[28] Danissa V. Rodriguez and Doris L. Carver. Comparison of information retrieval techniques for traceability link recovery. In *International Conference on Information and Computer Technologies*, 2019.

[29] G. Rothermel, R. Untch, C. Chu, , and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 179–188. IEEE-ACM, 1999.

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

[31] George Spanoudakis, Andrea Zisman, Elena Perez-Minana, and Paul Krause. Rule-based generation of requirements traceability relations. *The Journal of Systems and Software*.

[32] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *In Proceedings of 26th International Symposium on Software Testing and Analysis*, 2017.

[33] http://xml.apache.org/security.

[34] Annie T.T. Ying and Gail C. Murphy. Predicting Source Code Changes by Mining Change History. *IEEE RANSACTIONS ON SOFTWARE ENGINEERING*, September 2004.

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

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

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

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

[39] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *International Conference on Software Engineering*, pages 563–572. ACM, 2005.