Received 7 May 2023, accepted 30 May 2023, date of publication 5 June 2023, date of current version 13 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3283212

RESEARCH ARTICLE

Application of the Law of Minimum and Dissimilarity Analysis to Regression Test Case Prioritization

EKINCAN UFUKTEPE [1], (Member, IEEE), AND TUGKAN TUGLULAR [2], (Member, IEEE)
[1]Department of Electrical Engineering and Computer Science, University of Missouri, Columbia, MO 65201, USA
[2]Department of Computer Engineering, İzmir Institute of Technology, 35430 İzmir, Turkey

Corresponding author: Ekincan Ufuktepe (euh46@missouri.edu)

ABSTRACT Regression testing is one of the most expensive processes in testing. Prioritizing test cases in regression testing is critical for the goal of detecting the faults sooner within a large set of test cases. We propose a test case prioritization (TCP) technique for regression testing called *LoM-Score* inspired by the Law of Minimum (LoM) from biology. This technique calculates the impact probabilities of methods calculated by change impact analysis with forward slicing and orders test cases according to LoM. However, this ordering doesn't consider the possibility that consecutive test cases may be covering the same methods repeatedly. Thereby, such ordering can delay the time of revealing faults that exist in other methods. To solve this problem, we enhance the *LoM-Score* TCP technique with an adaptive approach, namely with a dissimilarity-based coordinate analysis approach. The dissimilarity-based coordinate analysis uses Jaccard Similarity for calculating the similarity coefficients between test cases in terms of covered methods and the enhanced technique called Dissimilarity-LoM-Score (*Dis-LoM-Score*) applies a penalty with respective on the ordered test cases. We performed our case study on 10 open-source Java projects from Defects4J, which is a dataset of real bugs and an infrastructure for controlled experiments provided for software engineering researchers. Then, we hand-seeded multiple mutants generated by Major, which is a mutation testing tool. Then we compared our TCP techniques *LoM-Score* and *Dis-LoM-Score* with the four traditional TCP techniques based on their Average Percentage of Faults Detected (APFD) results.

INDEX TERMS Change impact analysis, regression testing, software testing, test case prioritization.

I. INTRODUCTION

Nowadays software has a tendency to grow too fast and get excessively complex. This is because of the rapidly changing requests from users and technological advancements. For instance, Google has announced that approximately 50% of their code changes every month. In addition, they have mentioned that they commit 20 code changes per minute, and they run approximately 1 million test cases every day [1]. Furthermore, in web-based systems, considering the number of dependencies and related web services, the impact that is caused by changes can be drastic, and web-based systems tend to evolve faster compared to desktop applications [2], [3]. In such an environment, where codes are changing frequently, a huge amount of test cases makes the regression testing process a difficult task. Therefore, instead of waiting for the entire regression testing to be completed, studies have focused on prioritizing test cases in a test suite to find bugs earlier by giving high priority to test cases that are like to expose the existing bugs for the software under test (SUT). In other words, the test case prioritization (TCP) problem aims to find an execution order of available test cases in a test suite T to maximize a selected objective function. The formal definition of TCP problem [4] is given below:

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

Problem: *Find* $T' \in PT$ *such that:*

$$(\forall T'')(T'' \in PT), (T'' \neq T') \left[f(T') \geq f(T'')\right].$$

For the given TCP definition and problem, PT is the set of all possible permutations (orderings/prioritizations) of test

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

suite T, and f is the test case prioritization function that is applied to any such ordering, that returns a value, which represents the function's effectiveness. This function can be based on the coverage, mutant kill success, or APFD (Average Percentage of Faults Detected). The T' and T'' are each different orderings, where we are trying to find T', which has a higher score based on the applied f function. In this study, for the function f, we use APFD which is a metric to evaluate the given test order on how effective the order is for detecting faults earlier. The details and the formula for the APFD function are given and further discussed in Section V-C.

Focusing on changed methods is an effective approach to prioritize test cases based on the changed methods, or components. However, the existing test case prioritization techniques that use change information focus directly on the changed components, not on the other components that are dependent on the changed component. There are some components that could be affected by its dependent changed components, even though the affected component is not changed at all. Thereby, the changed components may not be affected by the changes that are made in themselves but might affect other components indirectly. Briefly, some faults cannot be detected by the test cases, which are only prioritized based on the changed methods. For instance, a method might be changed in order to adapt to a new standardization. However, its dependent methods are not changed with respect to the standardization change. Therefore, a fault appears not in the changed code, but in the unchanged methods. To detect such faults, dependencies should be checked with respect to the changed method. This study has focused on prioritizing test cases based on the changed methods and their affected methods that might be changed or unchanged.

In this study, we have proposed a method-level granularity test case prioritization technique that uses change impact analysis information. The change impact analysis information provides probabilistic information for each method that exists in the analyzed software. The probabilistic information represents information on methods being affected by the changes that are made in the software. Therefore, the higher probability the method has, the higher chance it is affected by the methods. The proposed change impact analysis uses three types of information; call graph of the latest software, method change information between two consecutive versions of software, and program slicing information. Among these three types of information, two of them are dependency-related information. The call graph information represents the calling dependencies between methods, which also shows the information flow between them. In addition, the program slicing information shows the possible affected statement with respect to a given data, i.e. slicing criterion. We use the method parameters for the slicing criteria. Return values from methods can also be used as a slicing criterion, however, it is found that including the return values tend to increase the false positives in change impact analysis [5]. In the end, we use call graph, method change, and program slicing information in a Markov chain model to calculate the posterior effects of changed methods and calculate the probability of being affected for each method in the software. After the Markov chain calculation and information about the changed methods, the probabilities of some methods may be increased or decreased. Two reasons can cause a decrease; the method is not changed or strongly affected by its dependent methods. Similar to the reasons for a decrease, two reasons can cause the probability increase; the method is changed, or its strongly dependent methods might be changed. Then, based on the calculated probabilities, we prioritize the test cases using two of our proposed test case prioritization techniques *LoM-Score* and *Dis-LoM-Score*, which are inspired by the Law of Minimum in agronomy and biology.

The main motivation of this study is that a single test case can cover more than one method, and the covered methods can be covered with different ratios compared to other test cases. Best to our knowledge, there isn't any study on using numerical change impact analysis results in test case prioritization. Using numerical change impact analysis values has its own difficulties. For instance, each test case does not cover the same amount of methods. Simple approaches such as calculating the mean of numerical values of covered impacted methods do not treat each test case fairly. Our empirical studies have shown that, while some test cases only cover 1-2 methods, others in the same project cover more than 100 methods. Using a mean calculation to determine the priority score of a test case can result in decreasing the priority of a test case that has a high potential of revealing faults. In this study, we have proposed an algorithm to determine a generic number called *lom*, which represents the number of how many impacted methods with the highest impact score that will be selected for a test case and calculate the average (mean) based on the *lom* number and selected impact scores. Furthermore, to enhance the fault detection rate, we have proposed another *LoM-based* approach called *Dis-LoM-Score* that follows an *Adaptive* approach that uses dissimilarity. The LoM approach can be adopted with another change impact analysis technique that provides numerical values on methods. We have used and enhanced our previously proposed change impact analysis technique for the numerical change impact analysis results of methods in the software.

Our case study results have shown that both *Dis-LoM-Score* and *LoM-Score* are more reliable TCP approaches compared to the other state-of-the-art approaches. Furthermore, we have found that the *Total-Diff* TCP approach [6] had competitive results with our *Dis-LoM-Score* and *LoM-Score* TCP approaches, but *Total-Diff* has also performed under Random TCP for a project in our case study, which doesn't make the *Total-Diff* TCP a robust or reliable TCP approach.

In this study, we want to answer two research questions in order to produce a better understanding of solving our problem.

RQ1: How can change impact analysis information be used in test case prioritization and is it effective? There are various studies on change impact analysis mentioning that change impact analysis can be used in test case prioritization [7], [8], [9]. However, there aren't any significant studies on how change impact analysis results can be applied on the test case prioritization problem. In this study, we investigate how probabilistic change impact analysis information can be applied in test case prioritization, and evaluate the effectiveness by comparing their APFD performances with other state-of-the-art TCP methods.

RQ2: Does using change impact analysis information solve the problem of similar test cases being ordered consecutively in test case prioritization? Most test case prioritization techniques suffer from prioritizing test cases that are similar, consecutively. This decreases the Average Percentage Faults Detected (APFD) [4] results of the test case prioritization techniques. Therefore, these test case prioritization techniques use "*Additional*" or *Adaptive* techniques as in [4] to increase the APFD results. We want to discover if using change impact analysis information requires any "*Additional*" or a *Similarity*-based approach to enhance its APFD results.

This paper makes the following main contributions:

(i) Method: We present two novel test case prioritization techniques (*LoM-Score* and *Dis-LoM-Score*), that use change impact analysis information and fuse the probabilistic change impact analysis result for each covered method in the source code to calculate a test case score that is used for prioritization. The fusion process of the probabilistic data for each covered method for each test case is inspired by the *Law of Minimum* from agronomy and biology. The difference between the proposed two test case prioritization techniques is to include the similarity information in one of them. Thereby, one of the test case prioritization techniques would give a higher chance to other test cases that are not covering the same methods.

The *LoM-Score* uses the lower quartile information of the number of covered methods of each test case. The lower quartile value will be used for determining how many of the covered methods will be selected to calculate the score of each test case. We follow this approach to treat each test case fairly, which is one of our main contributions since each test case can cover a different number of methods. Without this approach, the calculated test score is affected by outliers. Furthermore, with this approach, are also able to assign numerical values to test cases. For the *Dis-LoM-Score*, we follow the same approach as *LoM-Score*, but also include a similarity analysis to update test scores on every selection of the test case based on Jaccard similarity.

(ii) Source Codes: We developed two test case prioritization methods, which we call *LoM-Score* and *Dis-LoM-Score*. We also made our source codes publicly available[1] for reproducibility.

[1]https://github.com/ekincanufuktepe/lom-tcp

(iii) Dataset: We shared our dataset that contains:

- Method coverage matrices generated by GZoltar[2] [10]
- Projects with hand-seeded mutants.
- Change impact analysis reports from Code Change Sniffer [5].
- Test case information used for test case prioritization.

The manuscript is organized as follows. In Section II, the fundamentals of program slicing, and change impact analysis are explained. In Section III, we provide the details on the change impact analysis approach that is used in TCP. In Section IV, the fundamentals of the "*Law of Minimum*" is given, along with the proposed *LoM-Score* and *Dis-LoM-Score* TCP approaches. In Section V, the case study, and set up of work are given. In Section VI, we discuss our findings and results, of our case study, where we also answer our research questions. In section VII, a summary of related work on test case prioritization, which is mostly based on probabilistic graphical models and program slicing. Finally, section VIII concludes the paper and mentions the future work of our study.

II. FUNDAMENTALS

A. PROGRAM SLICING

Program slicing is a technique for reducing the number of statements that are required to be absorbed by the programmer [11]. Given a point of "interest," which is also known as a statement criterion in a program is described by a variable and a statement. A program slice gives all the statements that have contributed to the value of the variable at the point and eliminates the unnecessary statements.

The first program slicing technique and idea was first developed by Weiser [12] called Backward slicing, which is a static analysis-based approach. The ideology of Backward slicing is to assist developers in order to locate the parts of a given program that might contain a bug. On the other hand, in 1990 Horwitz et al. [13] proposed a different program slicing technique called Forward slicing. Unlike Backward slicing, Forward slicing focuses on predicting the parts of a given program that might be affected after a modification or change has been made.

Since this study uses change impact analysis data to prioritize test cases, it concentrates on parts of the program that could be affected after changes are made to the source codes of software. Every change that is made to the source code carries a potential risk of introducing a fault in the software. Thereby, this study uses Forward slicing to reveal the possible parts of the source code that might be affected.

B. CHANGE IMPACT ANALYSIS

Change Impact Analysis (CIA) is a field of software evolution and maintenance, that focuses on detecting the affected/impacted parts of the code after a change has been committed to the software. The aim of CIA is to aid software developers by showing the impacted parts of the code to

[2]https://github.com/GZoltar/gzoltar

reduce software maintenance time. In addition, CIA aids software testers by showing impacted parts of code to be tested with higher priority. Thereby, the time of revealing bugs related to the changes is reduced.

To measure the effectiveness of a proposed CIA technique, two metrics are used [14]; Recall and Precision. Both of these metrics are used for providing an understanding and measuring relevance based on the estimations. The recall is a metric that provides an understanding of the success of successful estimations. On the other hand, precision is the fraction of the estimated impacted components that are relevant. The equation for Precision is given in Equation (1) and the equation for Recall is given in Equation (2) In terms of software maintenance and CIA, the Recall metric is more important than the Precision [15] because the cost of missing an impacted would be more expensive, than checking all the methods if they are impacted or not.

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{1}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{2}$$

III. CHANGE IMPACT ANALYSIS USING FORWARD SLICING AND MARKOV CHAINS

For change impact analysis we used Code Change Sniffer [5], which uses Markov chains to calculate the probabilistic values for each method being impacted by changes. In this section, we will give brief information on how Code Change Sniffer [5] works, by dividing it into four important steps;

1) Call graph information extraction
2) Changed method information extraction
3) Forward slicing implementation
4) Change impact analysis with Markov chains.

A. CALL GRAPH INFORMATION EXTRACTION

To extract the call graph information of a given open-source project, we first receive the final version or the last commit of the project for change impact analysis and test case prioritization. The reason for selecting the final version of the analyzed project is to extract the current and up-to-date information about the project. The call graph is generated by an open-source tool called java-callgraph,[3] which is available on GitHub. The tool allows static and dynamic call graph generation of a given open-source Java project. Since this study has focused on a static approach, we have used the static call graph generation feature. In addition, the tool has been slightly modified to store the generated call graph in a Map data structure, where keys are the source nodes (caller methods) and the value is a list of destination nodes (callee methods). Otherwise, java-callgraph only provides a printed output of a call graph.

B. CHANGED METHOD INFORMATION EXTRACTION

Extracting the information of changed methods plays an important role in this study. Without the change information, the change impact analysis cannot be performed. In order to extract such knowledge, we require two versions of projects, which are the previous and the current versions. Thereby, we can find which methods are changed. However, finding which methods are changed is not a sufficient amount of information for the change impact analysis process. In order to quantify and calculate the probability of a method being impacted, we need to numerically represent changes as well.

Code Change Sniffer [5] and previous studies [16], [17], [18] have numerically represented the amount of change based on the number of changed bytecode instructions in a method/class and the total number of bytecode instructions in a method/class. Code Change Sniffer works in a method-level granularity, therefore, our proposed TCP algorithm is also based on method-level information. To find the changed methods and the amount of change for each method, we have used an open-source Java-based tool called reJ.[4] reJ is a Graphical User Interface (GUI)-based tool that receives two different versions of the same open-source Java project. The differences between projects are visually shown in bytecodes for each class. The motivation behind working at the bytecode level is, even if there is no change in high-level language source code, some software or projects will just change the Java versions. These changes can lead to instruction-level changes, even if the source code is not changed. However, such a change can lead to an impact on the software. Furthermore, high-level source code *diff* algorithms might only look into textual differences. For instance, if statement $x = x + 1$ has been changed to $x + +$, a high-level *diff* algorithm (such as GitHub), will show this as a change. However, at an instruction level, this will not be considered a change.

It is also important to mention that the tool reJ only provides visual change data (by lines) at the class level without any numerical values. However, this study follows a method-level granularity approach. For this reason, we have extended the reJ tool that extracts change information in method-level granularity with numerical information that represents the ratio of change for each method between the 0.0-1.0 range. A method that has a ratio of change value that is equal to 0 represents that there are no changes made to the method. On the other hand, if a method has a ratio of change equal to 1, then this method is newly implemented and did not exist in the previous project version. Therefore, if a method has a ratio of change greater than 0 and less than 1, then the method exists both in the previous and current versions and has been changed. The changes include adding, removing, and editing statements on the bytecode. To calculate the ratio of change we use Equation (3). In Equation (3), the $roc(m_i)$ is the function that calculates the *ratio of change* for method m_i

[3]Java Call graph - https://github.com/gousiosg/java-callgraph

[4]reJ - http://rejava.sourceforge.net/

that exist in the project.

$$roc\,(m_i) = \frac{\#\ of\ changed\ bytecode\ instructions}{\#\ of\ total\ bytecode\ instructions} \tag{3}$$

C. FORWARD SLICING IMPLEMENTATION

Program slicing has been used in several graphical models of programs such as control-flow graphs (CFG) [12], program-dependence graphs (PDG) [19], system-dependence graphs (SDG) [13]. This study has used CFGs to run the forward slicing technique, and in order to generate CFGs from Java source code we have used Soot[5] [20], [21]. Soot is a very popular framework among program slicing tools, for generating CFGs and applying program slicing techniques to them.

There are very few program slicing tools written for Java programming language. Jayaraman et al. [22] proposed a Java program slicer for Eclipse, which is called Kaveri.[6] It provides two static program slicing techniques, Backward and Forward slicing. Furthermore, for better understanding, it provides a visual output of the calculated slices. Venkatesh et al. [23] used and improved the Java program slicing tools Kaveri and Indus[7] in order to compute and visualize concurrent Java programs. On the other hand, there is another alternative open-source Java-based program slicing tool called WALA.[8] WALA does not only provide static analysis features for Java but also for JavaScript language as well. The basic static analysis features of WALA are; context-sensitive tabulation-based slicing, pointer analysis, call graph, and system-dependence graph construction, etc. However, in this study, we haven't used any of the existing program slicer tools, because they are not maintained anymore or did not support new versions of Java. Therefore, we have implemented the forward slicing technique in order to make it compatible with the libraries and Java versions that we have used in this study.

D. CHANGE IMPACT ANALYSIS WITH MARKOV CHAINS

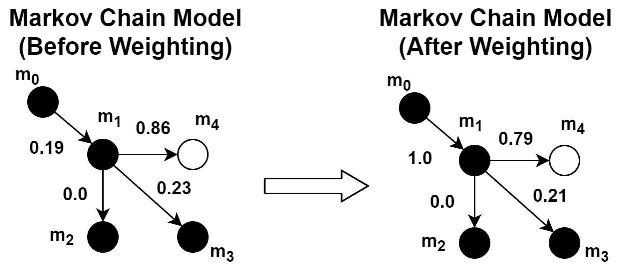
Code Change Sniffer [5] is a change impact analysis tool that uses the Markov chain to calculate the probabilities of impacted methods in the software. The probabilistic information is computed by analyzing the software with static analysis. The *diff* information between two commits (or versions) is used as an initial vector, while the transition matrix is filled with probabilistic information acquired from forward slicing. In the following paragraphs of this section, we provide a small running example of how Code Change Sniffer works.

The probabilistic information obtained from forward slicing is encoded into the Markov chain's edges along with the change information based on the type of the model, namely, call graph (CG) and effect graph (EG). However, it is important to mention that we used call graphs in this study to analyze the impacts. On the other hand, the initial vector is encoded with change information, which applies to both models. Starting with encoding the edges, we construct a transition matrix, which is similar to an adjacency matrix.

The forward slicing technique is applied to the method parameters, assuming the change impact will propagate from the caller to the callee method through parameters. Since the method parameters are defined in the first statements of the Control-flow graph (CFG), the forward slicing starts from the first statement to the last statements (leaf/sink nodes). After the sliced CFG is obtained, we calculate the probability of the impacted method. The impact probability is calculated by dividing the remaining statements in CFG after forward slicing by the total number of statements that exist in the CFG before slicing. We that each method has its own CFG, which is consisted of statements. Thereby, let CFG_{m_i} be a set of statements of method m_i and let $stm_{m_{ik}}$ be the statements in the CFG, where k is the statement id of method m_i. Then, in Equation 4, let $pCFG_{m_i}$ be the parameter-based sliced CFG of CFG_{m_i}, which is also a subset of CFG_{m_i}. The calculation of change probability for parameter-based forward slicing is given in Equation 5. For methods that do not have any parameters, the probability of change is set to 0.

$$CFG_{(m_i)} = \{stm_{m_{ik}} : 0 < k \leq |CFG_{m_i}|, m_i \in M\}$$
$$where\ pCFG_{m_i} \subseteq CFG_{m_i} \tag{4}$$
$$P(m_i) = \frac{|pCFG_{m_i}|}{|CFG_{m_i}|} \tag{5}$$

Another property of the Markov chain is that summation of the outgoing edge probabilities of a node should be equal to 1. Therefore, the probability summation of each row in the transition matrix should be equal to 1. However, a row summation could be less than or greater than 1 depending on the probabilities obtained from forward slicing. For instance, on the left side of Fig. 1, let us assume that we encode the Markov chain model with probabilistic information with forward slicing and change information. We can see that some of the nodes' summation of outgoing edges are less than 1 or greater than 1. To satisfy the properties of the Markov chain, we weight each node's outgoing edges, by dividing the summation of outgoing edges by each outgoing edge of that node. On the right-hand side of Fig. 1, we obtain the updated Markov chain after weighting the edges. Furthermore, assume that the filled methods (m_0, m_1, m_2, m_3, m_4) are the changed methods.



Markov Chain Model (Before Weighting)
m0
m1
m4
m2
m3
0.19
0.86
0.23
0.0
Markov Chain Model (After Weighting)
m0
m1
m4
m2
m3
1.0
0.79
0.21
0.0

FIGURE 1. Markov chain model construction with weighted edges.

[5]Soot Framework - https://github.com/Sable/soot

[6]Kaveri - http://indus.projects.cs.ksu.edu/projects/kaveri.shtml

[7]Indus - http://indus.projects.cs.ksu.edu/

[8]WALA - http://wala.sourceforge.net/

After the weighting process is completed, we construct the transition matrix of the Markov chain model below. According to the graph model in Fig. 1, there is no outgoing edge from methods m_2, m_3, and m_4. Therefore, in the transition matrix, we would expect to have the entire row filled with 0s. However, we have a single 1 that is placed to itself such as $m_2 \rightarrow m_2$, $m_3 \rightarrow m_3$, $m_4 \rightarrow m_4$. According to the Markov chain's properties, the summation of the columns for each row should be equal to 1. Thereby, for a row where the sum of column values is equal to 0, we set the $m_i \rightarrow m_i$ edge probability to 1. If the method m_i is not changed, setting the probability will not affect the overall impact calculation, since it will be multiplied with 0.

$$T = \begin{array}{c} \\ m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \end{array} \begin{array}{c} \begin{array}{ccccc} m_0 & m_1 & m_2 & m_3 & m_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.21 & 0.79 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

To calculate the impact vector, in other words, the vector that contains the probabilities of predicted methods that will change, an initial vector should be multiplied by the transition matrix. We encode the initial vector with change information we have collected from diff calculations. The change information represents the likelihood of a method that could affect itself by the changes that are made to the current method. Therefore, as the amount of change increases the probability of being affected by changes will be higher. In Fig. 1, let's assume the filled nodes (methods) m_0, m_1, m_2 and m_3 in the Markov chain are the changed methods with given probabilities; $m_0 = 0.5$, $m_1 = 0.71$, $m_2 = 0.78$ and $m_3 = 0.33$. The four given change probabilities are encoded into the initial vector below. Previously, to satisfy the properties of the Markov chain in the transition matrix, we weight the edges of each node's outgoing edges. Similarly, we also need to weight the initial vector values as well. According to the Markov chain's properties, the summation of the probabilities in the initial vector should be equal to 1, where the sum of the probabilities in our initial vector is greater than 1.

$$I = [0.5\ \ 0.71\ \ 0.78\ \ 0.33\ \ 0] \tag{6}$$

We weight the initial vector by dividing each value in the vector by the summation of the probabilities in the vector. Thereby, we have updated our initial vector I to I_w, which is given below.

$$I_w = [0.216\ \ 0.306\ \ 0.336\ \ 0.142\ \ 0] \tag{7}$$

Finally, we obtain the final forms of our initial vector and transition matrix, and by using the final forms of the initial vector and transition matrix, we calculate the impact vector in Equation (8), which is predicted to be changed methods. Since our initial vector and transition matrix is weighted, we expect to calculate the impact vector, where the summation of its probabilities is equal to 1.

$$I_w \cdot T = [0\ \ 0.216\ \ 0.336\ \ 0.206\ \ 0.242] \tag{8}$$

Based on the Markov chain model in Fig. 1 and calculation in Equation (8), the probabilities of the methods being affected by the changes are calculated as $m_0 = 0.0$, $m_1 = 0.216$, $m_2 = 0.336$, $m_3 = 0.206$, and $m_4 = 0.242$. With respect to the probabilities, m_2 is the method that has the highest likelihood of being affected by the changes.

IV. USING THE LAW OF MINIMUM (LoM) PRINCIPLES WITH CHANGE IMPACT ANALYSIS INFORMATION FOR TEST CASE PRIORITIZATION

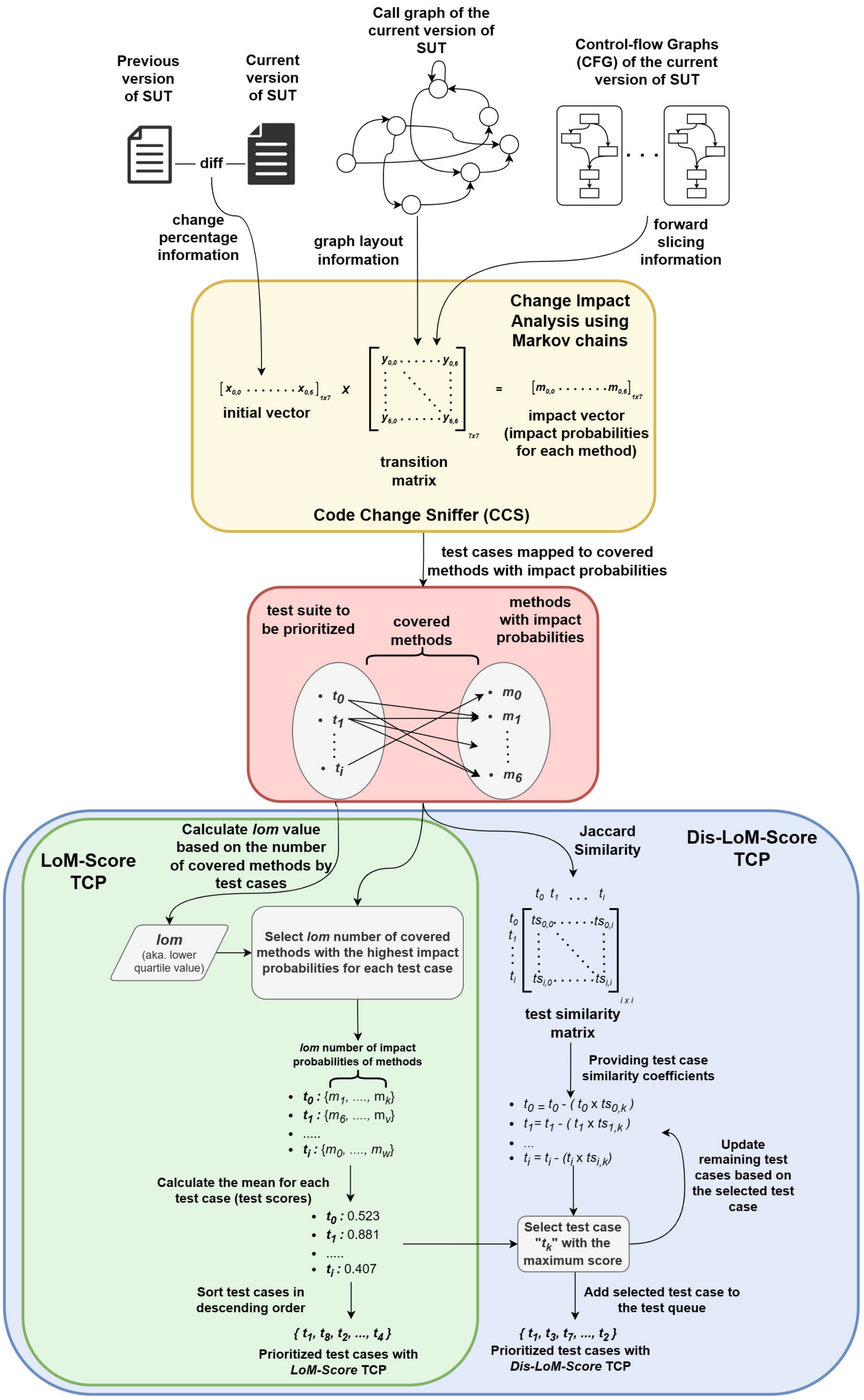
In this section, we first explain what the *Law of Minimum* is that has inspired our novel TCP approaches: *LoM-Score* and *Dis-LoM-Score*. Then, we provide the details on how we have applied the *Law of Minimum* principles in using change impact analysis for test case prioritization, which are the proposed test case prioritization approaches in this study. The first proposed test case prioritization is we propose is called *LoM-Score*, which calculates a score for each test case based on the *Law of Minimum* principles and change impact analysis information. Then, we explain our second proposed test case prioritization approach that is called *Dis-LoM-Score* (Dissimilarity Law of Minimum Score), which is similar to the *LoM-Score* TCP approach but takes into account the similarity of test cases based on the coverage while calculating the test case scores. In Figure 2, we provide the pipeline and architecture of the *LoM-Score* and *Dis-LoM-Score* TCP approaches we proposed.

A. THE LAW OF MINIMUM

The "*Law of the Minimum*" (LoM) aka. "*Liebig's Law of the Minimum*" is originally a concept that is used in biology, agricultural science, and nature. Examples of the *LoM* can be widely seen in the growth and development of plants and animals, and are determined by the availability of that essential nutrient, which is present in the smallest amount. There are various other examples of the "*Law of the Minimum*" as well. For instance, if one growth factor/nutrient is deficient, plant growth is limited, even if all other vital factors/nutrients (macronutrients) are adequate. A plant could also use other non-vital factors/nutrients (micronutrients), however, these factors/nutrients are does not have a vital effect on a plant's growth. Another simple example of the "*Law of the Minimum*" is, an elephant herd manages its speed based on the slowest elephant in the herd. In other words, the ideology of the "*Law of the Minimum*" basically means, "*A chain is only as strong as its weakest link.*"

B. LoM-SCORE TEST CASE PRIORITIZATION

As we mentioned in Section IV-A, there are vital factors/ nutrients (macronutrients), and there are other non-vital factors/nutrients (micronutrients) in the *LoM* principles. However, macronutrients play a key role in plant growth, and on



Previous version of SUT
Current version of SUT
diff
Call graph of the current version of SUT
Control-flow Graphs (CFG) of the current version of SUT
change percentage information
graph layout information
forward slicing information
Change Impact Analysis using Markov chains
initial vector
transition matrix
impact vector (impact probabilities for each method)
Code Change Sniffer (CCS)
test cases mapped to covered methods with impact probabilities
test suite to be prioritized
covered methods
methods with impact probabilities
LoM-Score TCP
Calculate lom value based on the number of covered methods by test cases
lom (aka. lower quartile value)
Select lom number of covered methods with the highest impact probabilities for each test case
lom number of impact probabilities of methods
Calculate the mean for each test case (test scores)
Sort test cases in descending order
Prioritized test cases with LoM-Score TCP
Dis-LoM-Score TCP
Jaccard Similarity
test similarity matrix
Providing test case similarity coefficients
Update remaining test cases based on the selected test case
Select test case "t_k" with the maximum score
Add selected test case to the test queue
Prioritized test cases with Dis-LoM-Score TCP

FIGURE 2. The pipeline architecture of the *LoM-Score* and *Dis-LoM-Score* TCP approaches.

the other hand, micronutrients do not have an effect on plant growth. Therefore, we can consider the macronutrients as the data that are within the interquartile range, while the micronutrients are the data outside the interquartile range and outliers.

We also recall that based on the *LoM* principles, a plant's growth is based on the minimum macronutrient among the other macronutrients, which indicates the lower quartile of the interquartile range. In other words, the minimum value of the interquartile range.

In the context of test case prioritization and change impact analysis, the number covered and impacted methods by a test case corresponds to the available nutrients in the plant. We determine the number of covered impacted methods that are greater than 0, which were calculated based on the change impact analysis. After we calculate the total number of covered impacted methods for each test case we find the lower quartile (aka. first quartile, and $Q1$), which we will be referring to as the *lom* (aka. *LoM-Score*). The *lom* is the median/middle value of the first half of the rank-ordered dataset, which is the same Equation (9) used for calculating the lower quartile value ($Q1$). In Equation (9), n is the size of the rank-ordered dataset. In Figure 3, we provide an example of how the lower quartile *Q1* (*lom*) is calculated. In Figure 3, assume that there are 10 test cases, along with the number of methods they cover in ascending order. We first split the list of the number of covered methods by each test case by half. To calculate the *lom* (lower quartile Q1) score, we find the median of the lower half, which is 2. Therefore, the *lom* score is calculated as 2.

$$lom = \left(\frac{n+1}{4}\right)^{th} term \tag{9}$$

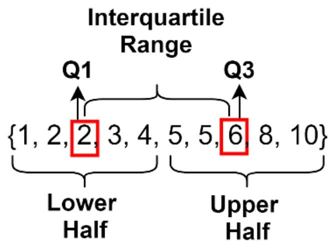


FIGURE 3. Example on finding the lower quartile Q1 (aka. *lom*).

Once the *lom* is calculated, this determines how many of the impacted methods will be selected to calculate the test case score. The selection process will pick *lom* number of the highest impact probability values that are covered for each test case. Scenarios where the number of covered methods by the test case is lower than *lom*, will add the value 0 until it reaches the size of *lom*. Once every test case has selected the highest impact probabilities and reached the set size of *lom*, we calculate the mean of the selected impact probabilities for each test case to assign a test case score. After the scores for each test case are calculated, the test cases are sorted in descending order, which is the prioritized test case. Test cases that share the same test scores are selected randomly. In Algorithm 1, we provide a detailed algorithm on how the *LoM-Score* TCP works.

Algorithm 1 LoM-Score Test Case Prioritization Algorithm

```
Input: $D$ is the test case coverage probability
        dictionary, where the key is the test case, and
        value is the set of impacted covered method
        probabilities ($D_{key} \leftarrow t$, and $D_{value} \leftarrow I$)
Input: $T$ is the set of test cases
Output: $T'$ is the set of prioritized test cases
/* let $C$ be a set of the number of covered
   impacted methods for each test case */
$C \leftarrow \emptyset$
// initialize set $C$
for $t_k$ in $T$ do
    // get the impacted covered method set of
       test case $t_k$
    $I_k \leftarrow D(t_k)$
    $C \leftarrow C \cup |I_k|$
end for
// initialize LoM-Score value
$lom \leftarrow getLowerQuartile(C)$
let $S$ be a dictionary, where the test case is the key,
  and the test score is the value
$S \leftarrow \emptyset$
for $t_k$ in $T$ do
    $I_k \leftarrow D(t_k)$
    while $|I_k| < lom$ do
        /* add the element 0 to set $I_k$ until it
           has $lom$ size of elements */
        $I_k \leftarrow I_k \cup \{0.0\}$
    end while
    let $I_k' \subseteq I_k$
    $I_k' \leftarrow getHighestLoMElements(I_k, lom)$
    $s \leftarrow 0$ // $s$ is the total score for test $t_k$
    for $i_j$ in $I_k'$ do
        /* add all the covered impacted method
           probabilities in $I_k'$ */
        $s \leftarrow s + i_j$
    end for
    // calculate the average of covered
       impacted methods
    $S(t_k) \leftarrow \frac{s}{lom}$
end for
// prioritize test cases in descending order
$T' \leftarrow sortTestCases(S)$
```

C. DIS-LoM-SCORE TEST CASE PRIORITIZATION

The *Dis-LoM-Score* test case prioritization we propose is based on the *LoM-Score* test case prioritization approach. The difference is that the *Dis-LoM-Score* also includes the dissimilarity between test cases. We recall that state-of-the-art test case prioritization techniques such as *Total*, and *Total-Diff* (details given in Section V-B), have alternate test case prioritization techniques. For instance, the alternate TCP for *Total* is *Additional*, and for *Total-Diff* is *Additional-Diff*. These techniques aim for prioritizing test cases that are more likely to be different than the test cases that have high priority, and this is done by giving high priority to test cases that have a higher coverage that is not covered yet. Since that our proposed *LoM-Score* TCP technique generates a score for test cases we couldn't incorporate the coverage information directly. Therefore, we used the Jaccard similarity between test cases based on the methods they covered. In Equation (10), the formula for calculating the Jaccard similarities between test cases are given, where $coverage(t_i)$ is the set of the covered method by test t_i. To calculate the similarities between the two test cases, the intersection of covered methods is divided by the union of covered methods. The similarity percentages that are calculated for each test pair will be used in every iteration during the prioritization to update test scores.

$$J(t_i, t_j) = \frac{|coverage(t_i) \cap coverage(t_j)|}{|coverage(t_i) \cup coverage(t_j)|} \tag{10}$$

In Algorithm 2, we provide the algorithm for our proposed *Dis-LoM-Score* TCP approach. The *Dis-LoM-Score* TCP approach uses the calculated test scores that are calculated from the *LoM-Score*. First, the test case with the maximum score will be selected as the highest-priority test case, which corresponds to l in Equation (11), and L is the set of test scores calculated by *LoM-Score*. Based on the selected test case, every test case score is updated by multiplying with the coefficient calculated by $J(l', l)$), and then subtracted by the test score.

$$\begin{aligned} l &\in L \\ L' &\leftarrow L - \{l\} \\ \forall l' &\in L', l' \leftarrow l' - (l' \cdot J(l', l)) \\ J(l', l) &\in [1, 0] \end{aligned} \tag{11}$$

Algorithm 2 Dis-LoM-Score Test Case Prioritization Algorithm

```
Input: $D$ is the test case coverage probability
        dictionary, where the key is the test case, and
        value is the set of impacted covered method
        probabilities ($D_{key} \leftarrow t$, and $D_{value} \leftarrow I$)
Input: $T$ is the set of test cases
Output: $T'$ is the set of prioritized test cases
let $S$ be a dictionary, where the test case is the key,
  and the test score is the value
// initialize the prioritized test cases
$T' \leftarrow \emptyset$
// collect test scores from LoM
$S \leftarrow getLoMTestScores(D, T)$
while $T \neq \emptyset$ do
    $t_k \leftarrow getTestWithMaxScore(S)$
    for $t_i$ in $T$ do
        if $t_k \neq t_i$ then
            $X \leftarrow intersectionOfCoverage(t_i, t_k)$
            $Y \leftarrow unionOfCoverage(t_i, t_k)$
            // calculate the Jaccard similarity
            $j \leftarrow \frac{|X|}{|Y|}$
            /* update the test scores of the
               remaining tests cases based on
               the selected test case          */
            $S(t_i) \leftarrow S(t_i) - (S(t_i) * j)$
        end if
    end for
    $T \leftarrow T - \{t_k\}$
    $T' \leftarrow T' \cup \{t_k\}$
end while
```

V. CASE STUDY

A. PROJECT SELECTION AND MUTATION TESTING

In our case study, we have selected and used 10 Java projects from Defects4J [24], which is given in Table 1. Defects4J is a framework that allows researchers to experiment with real bugs and fixes. In our case study, we couldn't use the real bugs from Defects4J, since each bug is isolated in a separate commit, which makes it difficult to measure the APFD scores. The APFD score is meaningful when there are multiple faults included in the project so that test cases can be prioritized. If there is only one fault in the project, then there is no reason to prioritize the test cases and just find the test case that exposes the fault.

Paterson et al. [25], combined multiple faults from Defects4J so that test case prioritization techniques can be evaluated on multiple real faults. However, their study was focused on coverage-based test case prioritization techniques, where only one version of the project is sufficient. Our test case prioritization technique is based on changes, which requires 2 versions of the project. In the study of Paterson et al. [25], selecting the base project is another difficult problem, due to the projects being combined from multiple commits. Therefore, we have used Major [26], [27] a mutation tool on one of the fix versions from Dejects4J, and injected/seeded multiple mutants into the project.

There are also other notable state-of-the-art mutation frameworks such as Pitest [28], which is also meant for Java projects. Pitest has strengths in generating stable mutations and has capabilities of minimizing the number of equivalent mutants [29]. However, since we are using the APFD metric

TABLE 1. Studied project information from Defect4J.

Project	SLOC	# of covered methods	# of test cases
jsoup	3787	339	137
gson	17474	553	716
commons-csv	2074	73	52
jackson-databind	68160	3180	1115
jackson-xml	9040	395	160
commons-compress	8667	200	44
joda-time	82552	1086	3953
commons-codec	7051	100	100
jackson-core	21907	911	195
commons-lang	59673	236	489

in our study, we only focus on the killed mutants and not the mutation score. Therefore, equivalent mutants are not an issue in this study. The main reason we haven't used Pitest is that it does not generate the source codes of the generated mutants, since it works at a compile level. Furthermore, Major is integrated with Defects4J and its projects, which makes it easier to perform mutation testing.

Another reason why we used mutation faults is that studies have shown mutation faults can be a representative of real faults and it is appropriate to use mutation faults in the context of regression testing techniques [30], [31]. Nevertheless, Do and Rothermel [32] have empirically investigated the effectiveness of using mutation fault. On the other hand, MuJava provides various types of mutants in method [33] and class levels [34]. However, since this study proposes a method-level granularity approach, only method-level mutants have been selected and added to the selected open-source projects. Just et al. [35] did a study on if mutants are valid substitutes for real faults, and their results have shown that conditional operator replacement, relational operator replacement, and statement deletion mutants are more often coupled to real faults than other mutants, which is heavily considered during our mutant selection and seeding process.

In Table 2, we present the number of mutants that are generated by the Major mutation framework, along with the number of killed mutants. Since we are interested in mutants that are detectable we select our mutants among the killed mutants. However, there are some mutants that are applied to some statements and operators. Therefore, we had to select only one of them, which also reduces the number of mutants we can hand seed into the projects.

TABLE 2. Total number of the generated mutants by Major, with the number of killed mutants, and the number of selected and seeded mutants for each project.

Project	# of generated mutants	# of killed mutants	# of seeded mutants
jsoup	25	15	9
gson	37	19	12
commons-csv	98	50	13
jackson-databind	116	60	30
jackson-xml	342	121	53
commons-compress	395	139	50
joda-time	415	333	93
commons-codec	589	438	81
jackson-core	925	485	96
commons-lang	941	647	138

The Major mutation framework is capable of generating 9 mutant types. However, we were only able to include 7 of the mutant types, which is given in Table 3 along with their brief descriptions. There are several reasons why we couldn't include all the mutants in our case study. One of them is due to the context of the projects that do not meet the requirements for performing a specific mutation type. For instance, shifting operations are commonly seen in embedded programming. However, the projects that we have studied are not mainly meant for embedded applications. Another reason is that, even though some mutant types were generated, they were not killed or detected by any test case.

Finally, in Table 4 we provide the details of the number of included mutant types for each project. During our hand-seeding process with mutants, we have tried to include as many mutants with different types. However, due to the number of killed mutants, the variety of mutants, and the overlap between different mutants, a significant amount of mutants were not included. For the reproducibility of our case study, we have also shared 10 of the hand-seeded projects from Defects4J.

In Table 4, we see that some of the mutant types were not as much as the others, such as LOR and ORU. There are two possible reasons why we couldn't include more of these mutant types:

- The operators to be mutated were not available or present in the projects. Therefore, these types of mutants were absent. Furthermore, the LOR and ORV mutants are not common mutants, because the mutation that is applied to these operators is mostly application dependent. For instance, shifting is a logical operator, and shifting is not a common operator. We would be able to see these operators mostly in embedding computing, or cryptography applications. Thereby, it is very unlikely to see these operators in software that we use in our daily lives.
- The mutant can be present, but the mutants were not killed. We cannot include mutants that survived, because they cannot be detected by any test case, which will affect the APFD score, and against the idea of the APFD calculation.

B. COMPARED TEST CASE PRIORITIZATION TECHNIQUES

In previous studies on TCP, it is quite common to see specific TCP techniques that are used for comparison with their proposed TCP techniques in their empirical studies. These techniques were proposed by Elbaum et al. [36], [37], and Do et al. [6]. They have proposed many techniques by following two basic approaches that are applied on different levels of granularity and different coverage criteria. These two approaches are *Total* and *Additional*. The *Total* approach gives higher priority and sorts test cases based on satisfying the maximum criteria to the minimum. The criteria can be the number of covered components, the probability of exposing fault, etc. On the other hand, the *Additional* approach follows a similar idea to *Total*. The *Additional* approach prioritizes and sorts the test cases based on the maximum components that are not covered yet.

Since our proposed TCP technique is a method-level granularity approach, to make a meaningful comparison we have selected five of the TCP techniques that are based on method-level granularity:

TABLE 3. Included mutant type information from Major.

Mutant Type	Acronym	Mutant Description
Arithmetic Operator Replacement	AOR	Replaces binary arithmetic operations for either integer or floating-point arithmetic with another op.
Conditional Operator Replacement	COR	Replace binary conditional operators with other binary conditional operators
Logical Operator Replacement	LOR	Replace binary logical operators with other binary logical operators
Literal Value Replacement	LVR	Replaces a literal with a default value
Operator Replacement Unary	ORU	Replaces a unary operator with another
Relational Operator Replacement	ROR	Replace relational operators with other relational operators
STatement Deletion	STD	Deletes (omits) a single statement

TABLE 4. Number of included mutants by mutant types.

Project	AOR	COR	LOR	LVR	ORU	ROR	STD
jsoup	-	1	-	-	1	7	
gson	-	3	-	3	-	2	4
commons-csv	1	1	-	4	-	3	4
jackson-databind	-	4	-	-	-	11	15
jackson-xml	-	10	-	13	-	5	25
commons-compress	6	2	1	15	-	10	16
joda-time	12	4	-	30	-	24	23
commons-codec	13	14	-	15	-	17	22
jackson-core	17	4	-	18	1	30	26
commons-lang	10	25	-	34	2	40	27

1) **Random Ordering TCP:** Random ordering is a technique that is used for experimental control, for setting the lower bound in the case study. Proposed TCP methods that perform under random are considered insignificant approaches. Basically, the random ordering shuffles the order of test cases in the test suite. In this study, 50 different random test case orders are generated and the average of their APFD results is calculated.
2) **Total TCP Technique:** Total [36], [37] is a coverage-based approach that orders test cases from the maximum number of covered methods to the minimum number of covered methods. This approach ignores if a method is already covered by prior test cases. If multiple test cases cover the same number of methods the selection for the order will be random.
3) **Additional TCP Technique:** Additional [36], [37] is another coverage-based TCP approach, which is very similar to the Total TCP technique. The only difference is that the Additional approach prioritizes the test cases based on a maximum number of methods that are not covered yet. Unlike the Total TCP technique, the Additional approach gives a higher chance to test cases that cover more methods that are not covered yet. If there are test cases that cover the same number of methods that are not covered yet, then a test case is selected randomly among the test cases.
4) **Total-Diff TCP Technique:** The main idea of the Total-Diff approach [6] is to prioritize by selecting the test cases that cover most of the changed methods. It ignores the methods that are not changed and focus on the amount of covered changed methods. Therefore, the test case that covers the most changed methods has a high priority. Similar to the Total TCP technique, the Total-Diff technique ignores if more than one test case is covering the same changed method. If multiple test cases cover the same number of changed methods the selection for the order will be random.
5) **Additional-Diff TCP Technique:** The Additional-Diff TCP technique [6] selects its test cases that cover most of the changed methods that are not covered yet. If more than one test case covers the same amount of change methods that are not covered yet, then the test case is selected randomly for the test case prioritization order. However, on each selection of the test case, the coverage information is updated for the remaining test cases.

C. EVALUATION MEASURE

In terms of fault detection, to measure the efficiency of the proposed test case prioritization techniques, the Average Percentage Fault Detection (APFD) metric is used. In addition, APFD is a measure that is widely used in test case prioritization studies. The APFD measures how soon the faults are detected based on the test case order. Higher values of APFD indicate faster fault detection. To formally define the APFD measure, let T be the set of n test cases and TF_i be the index of the first test case that detects the i^{th} fault. Also, assume that the number of faults/mutants that can be detected by the test suite is m, then the value of APFD metric of an ordering is given by following Equation (12) [4]. Based on Equation (12), the APFD value will range between 0 and 1, where the value of 1 indicates that all faults are detected by the first test case.

$$APFD = 1 - \frac{TF_1 + \cdots + TF_m}{nm} + \frac{1}{2n} \tag{12}$$

VI. RESULTS AND DISCUSSION

A. EVALUATION OF RESULTS

In this section, we first explain and compare experimental results and then investigate the statistical significance of the difference between our proposed TCP approaches *LoM-Score* and *Dis-LoM-Score* with the state-of-the-art TCP approaches. In total we perform three statistical tests, first, we perform the Tukey Honest Significant Difference (HSD) test to investigate the significance of TCP approaches for each project.

TABLE 5. APFD and ranking results for 10 projects from Defects4J.

commons-codec			commons-compress			commons-csv		
TCP Method	**Mean**	**Rank**	**TCP Method**	**Mean**	**Rank**	**TCP Method**	**Mean**	**Rank**
LoM-Score	0.9199	A	Total	0.9886	A	Dis-LoM-Score	0.8409	A
Total-Diff	0.9121	A	Additional	0.9886	A	Total-Diff	0.8288	A
Additional-Diff	0.8301	B	Dis-LoM-Score	0.9827	A	Additional	0.7871	B
Dis-LoM-Score	0.8241	B	LoM-Score	0.9823	A	LoM-Score	0.7092	C
Random	0.7802	C	Total-Diff	0.9818	A	Additional-Diff	0.6499	D
Additional	0.7749	C	Additional-Diff	0.7277	B	Random	0.6239	D
Total	0.6383	D	Random	0.6148	C	Total	0.4329	E

commons-lang			gson			jackson-core		
TCP Method	**Mean**	**Rank**	**TCP Method**	**Mean**	**Rank**	**TCP Method**	**Mean**	**Rank**
Total-Diff	0.9674	A	LoM-Score	0.9990	A	Dis-LoM-Score	0.9436	A
LoM-Score	0.9522	A	Total-Diff	0.9990	A	Additional-Diff	0.9181	B
Dis-LoM-Score	0.9134	B	Dis-LoM-Score	0.9988	A	Total-Diff	0.8705	C
Total	0.8283	C	Total	0.9988	A	LoM-Score	0.8697	C
Additional-Diff	0.8191	C	Additional-Diff	0.9901	A	Additional	0.7650	D
Additional	0.7355	D	Additional	0.9859	A	Random	0.7256	E
Random	0.5913	E	Random	0.7505	B	Total	0.5981	F

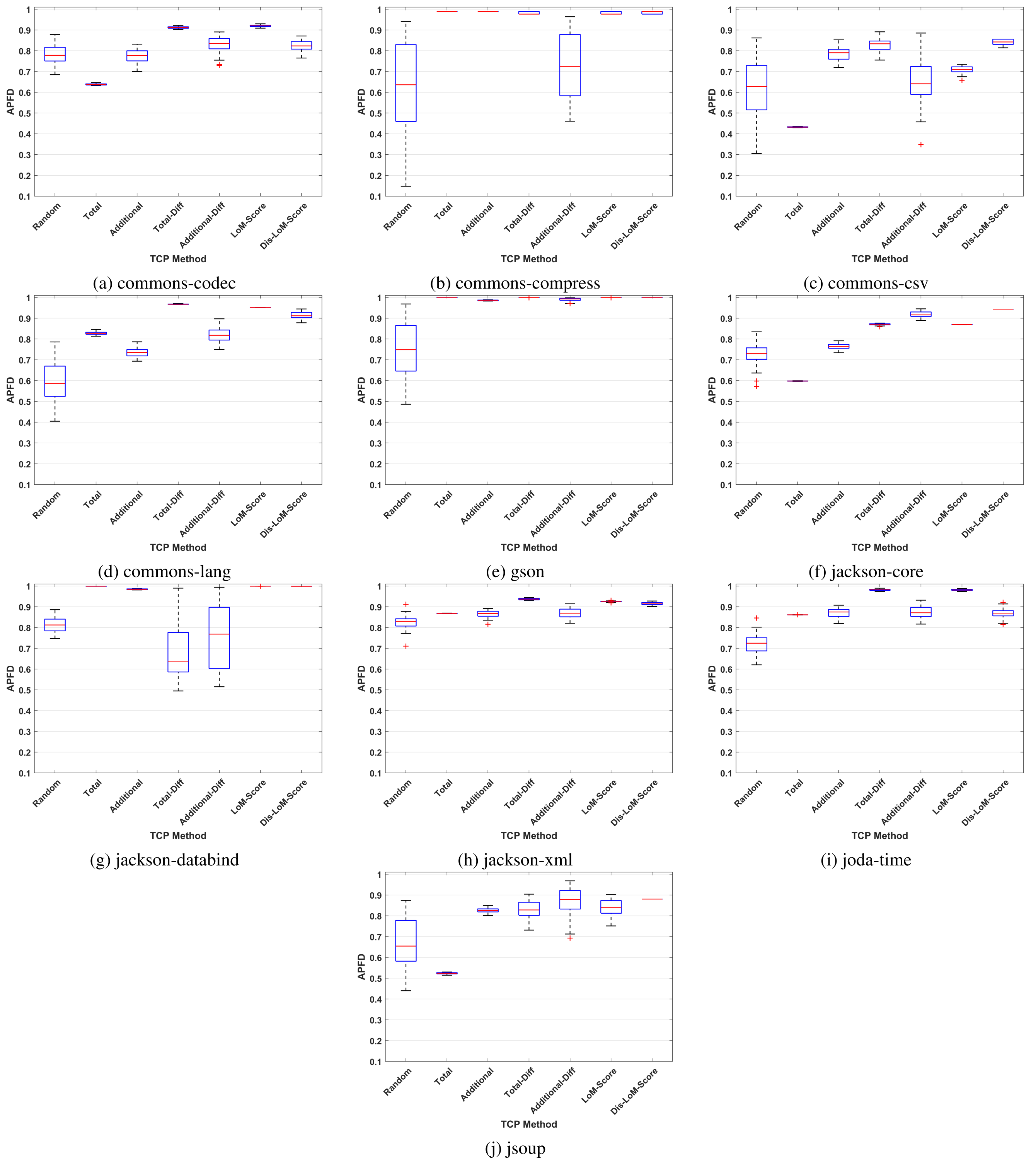
jackson-databind			jackson-xml		
TCP Method	**Mean**	**Rank**	**TCP Method**	**Mean**	**Rank**
LoM-Score	0.9990	A	Total-Diff	0.9371	A
Total	0.9988	A	LoM-Score	0.9248	B
Dis-LoM-Score	0.9987	A	Dis-LoM-Score	0.9163	B
Additional	0.9844	A	Total	0.8681	C
Random	0.8120	B	Additional-Diff	0.8677	C
Additional-Diff	0.7559	C	Additional	0.8654	C
Total-Diff	0.6815	D	Random	0.8267	D

joda-time			jsoup		
TCP Method	**Mean**	**Rank**	**TCP Method**	**Mean**	**Rank**
Total-Diff	0.9815	A	Dis-LoM-Score	0.8804	A
LoM-Score	0.9810	A	Additional-Diff	0.8714	A
Additional-Diff	0.8717	B	LoM-Score	0.8384	B
Additional	0.8713	B	Total-Diff	0.8300	B
Dis-LoM-Score	0.8670	B	Additional	0.8252	B
Total	0.8612	C	Random	0.6653	C
Random	0.7222	D	Total	0.5237	D

Then, we perform a Friedman test followed by the Wilcoxon sign test on the APFD results in 10 projects for each method. For all of our statistical tests, we used a statistical analysis software called IBM SPSS (Statistical Product and Service Solutions). These statistical tests are selected due to the observations on the TCP approaches not being independent of each other since they are run on the same open-source projects we have selected from Defects4J.

The ranks for each TCP method and for each project from the Tukey HSD post-hoc test are given in Table 5. The Tukey HSD post-hoc test is used for ranking and finding which TCP methods are significantly different from the others, where the ranks A, B, C, D, E, and F indicate the best to worst performing TCP method, respectively. The TCP methods that share the same rank indicate that the TCP methods are not significantly different. The *LoM-Score* TCP method has ranked A in 6 projects, B in 2 projects, and C in 2 projects. For *Dis-LoM-Score* TCP method has ranked A in 6 projects, and B in 4 projects. Since there was no significant difference found between the *Tota-Diff*, *Dis-LoM-Score*, and *LoM-Score* TCP methods we look into the APFD performance details of the *Tota-Diff* TCP method. We found that the *Tota-Diff* TCP method has ranked A in 7 projects, B in 1 project, C in 1 project, and D in 1 project. However, it is important to mention for project *jackson-databind* the *Tota-Diff* TCP method has been performed under the *Random*, and *Random* is used for setting the lower boundary for evaluating proposed TCP method. In other words, proposed TCP methods that perform under Random are considered to be unreliable and insufficient. Even though the *Total-Diff* TCP method has not shown any significant difference between *Dis-LoM-Score* and *LoM-Score*, our proposed TCP method has shown reliable, and consistent APFD results compare to *Total-Diff*.

Figure 4 shows box plots of the APFD results of 50 executions for 10 projects from Defects4J. In Figure 4, the leftmost TCP method is the Random approach, which is used for finding the lower boundary. If a TCP method performs under Random, it is considered an insignificant approach,



(a) commons-codec (b) commons-compress (c) commons-csv (d) commons-lang (e) gson (f) jackson-core (g) jackson-databind (h) jackson-xml (i) joda-time (j) jsoup

FIGURE 4. **Box-plot of APFD results for 10 open-source Java projects for 7 TCP methods.**

which we see that in *Total* TCP approach has performed under Random in *commons-codec* (Figure4a), *commons-csv* (Figure4c), *jackson-core* (Figure 4f), and *jsoup* (Figure 4j). For *Additional* TCP approach, we see it performs under Random in *commons-codec* (Figure 4a). For *Additional-Diff* TCP approach, we see it performs under Random in *jackson-databind* (Figure 4g). Finally, for *Total-Diff* TCP approach, we see it performs under Random in *jackson-databind* (Figure 4g). This indicates that every TCP method we have compared with has performed under Random for at least one project in our case studies, while none of our proposed TCP approaches performed under Random. These results also show that our proposed TCP methods are more reliable compared to the state-of-the-art TCP methods. Furthermore, in Figure 4, we observe that in some projects, the compared TCP methods that perform under Random can generate various test orders that result in sporadic APFD results. In other words, these TCP methods will have outliers, sometimes they can perform the best, and sometimes they can perform the worst for the same project. Therefore, in Figure 4 for some TCP methods and some projects there are wider box plots. On the other hand, our proposed TCP methods have narrow box plots, with very few to no outliers.

TABLE 6. (a) Ranking result of Friedman test on all 7 TCP techniques, (b) Ranking result of Friedman test on all TCP techniques except *Dis-LoM-Score*, (c) Ranking result of Friedman test on all TCP techniques except *Dis-LoM-Score* and *LoM-Score*.

TCP Technique	(a) Mean Rank	(b) Mean Rank	(c) Mean Rank
Random	1.70	1.70	1.70
Total	3.05	2.85	2.75
Additional	3.55	3.35	3.05
Total-Diff	5.30	4.80	4.30
Additional-Diff	3.50	3.30	3.20
LoM-Score	5.60	5.00	-
Dis-LoM-Score	5.30	-	-

The Friedman test is a non-parametric test that allows us to statistically test the difference between the results of all the approaches at once. In Table 6, column (a), we performed the Friedman test on all seven TCP approaches to investigate if there is a significant difference when all the TCP approaches are evaluated together. Then, in Table 6, column (b), we have excluded the *Dis-LoM-Score* TCP results from the Friedman test to observe if there is any significant difference between the remaining six TCP approaches. We then perform our third Friedman test which we excluded the *Dis-LoM-Score* and the *LoM-Score* TCP approaches and performed on the remaining five TCP approaches to see if there is any significant difference between the remaining approaches.

In Table 6, we also notice that for the Random approach, we have the same Mean Rank values on the three different rankings (a), (b), and (c). When the Friedman test is performed for each ranking (a), (b), and (c), we obtained the same value, i.e., 1.70, for the Random approach. Since the TCP approaches are ranked in ascending order for each ranking, the Random approach will always have the same lowest ranking.

In addition to the three Friedman test given in Table 6, we provide the details of the test statistics from the three Friedman tests in Table 7. The test statistics that are given in Table 7 show that in column (a), where both *Dis-LoM-Score* and *LoM-Score* TCP approaches are included in the Friedman test, with a significance level of 0.05, the p-value is $0.001 < 0.05$. Thereby, we can say that there is a significant difference between the TCP approaches in column (a). In column (b), where the *Dis-LoM-Score* is

TABLE 7. (a) Test statistics of Friedman on all 7 TCP techniques, (b) Test statistics of Friedman on all TCP techniques except *Dis-LoM-Score*, (c) Test statistics of Friedman on all TCP techniques except *Dis-LoM-Score* and *LoM-Score*.

Test Statistics	(a)	(b)	(c)
N	10	10	10
Chi-Square	27.016	21.963	14.010
df	6	5	4
p-value	0.001	0.001	0.007

excluded, we again see that the p-value is $0.001 < 0.05$. In column (c), where we excluded both *Dis-LoM-Score* and *LoM-Score* TCP approaches, we see that the p-value is $0.007 < 0.05$, which still indicates that there is still a significant difference between the remaining TCP approaches. However, we see a slight increase in the p-value.

After our observations, we see that there is a significant difference between the TCP approaches when we have the *Dis-LoM-Score* and *LoM-Score* TCP approaches in the Friedman test, which allows us to perform a Wilcoxon Sign test so that it can provide a detailed analysis. In other words, the Wilcoxon Sign test will show which TCP approaches have a significant difference when compared with our proposed *Dis-LoM-Score* and *LoM-Score* TCP approaches. We first perform the Wilcoxon Sign test between the *Dis-LoM-Score* TCP approach with the rest of all TCP approaches. Then, we perform the Wilcoxon Sign test between the *LoM-Score* TCP approach with the rest of all TCP approaches. First, in Table 8 we present the two-tailed Wilcoxon Sign test results of the *Dis-LoM-Score* TCP approach, and in Table 9 we present the two-tailed Wilcoxon Sign test results of the *LoM-Score* TCP approach. The given Z values in Tables 8 and 9 are calculated based on the positive ranks. The p-values are examined with a significance level of 0.05, where the p-values that are smaller than 0.05 indicate a significant difference between the TCP approaches.

TABLE 8. Significantly different methods compared to the *Dis-LoM-Score* method with respect to two-tailed Wilcoxon test.

TCP Method	Z	*p*-value
Random	-2.805	0.005
Total	-2.193	0.028
Additional	-2.499	0.012
Total-Diff	-0.255	0.799
Additional-Diff	-2.499	0.012
LoM-Score	-0.051	0.959

Based on the results in Table 8, we observe a significant difference between the *Dis-LoM-Score* TCP approach with *Random*, *Total*, *Additional*, *Additional-Diff* for a significance level of 0.05.

Based on the results in Table 9, we observe a significant difference between the *LoM-Score* TCP approach with *Random*, *Total*, *Additional-Diff* for a significance level of 0.05.

TABLE 9. Significantly different methods compared to the *LoM-Score* method with respect to two-tailed Wilcoxon test.

TCP Method	Z	p-value
Random	-2.805	0.005
Total	-2.499	0.012
Additional	-1.377	0.169
Total-Diff	-.561	0.575
Additional-Diff	-2.601	0.009

In addition to our Friedman and two-tailed Wilcoxon Sign test, we also performed a one-way Analysis of Variance (ANOVA) analysis on the mean APFD values for each project to statistically analyze the difference between the TCP methods. Then, we performed a Tukey HSD test on each project again on the mean of APFD values for seven TCP methods. The Tukey HSD post-hoc test categorizes the TCP methods into different groups/ranks with respect to their APFD performances. We considered the significance level $\alpha = 0.05$ for both of the statistical procedures.

B. ANSWERING RESEARCH QUESTIONS

RQ1: How can change impact analysis information be used in test case prioritization and is it effective? In this study, by using program slicing, call graphs, change information, and Markov chains, we were able to assign numerical values to the impacted methods. These impacted methods are represented with probabilistic information. The impacted method with a lower probability means that the methods can be affected by the changes with a lower chance. On the other hand, a method with a higher probability represents that, with a high chance the method might be affected by the changes that are made in the software and must be tested with higher priority. However, the problem in using these numerical values is that every test case has a different number of covering methods. Therefore, calculating the average of the numerical values of the test cases are not providing a fair evaluation while prioritizing the test cases. Using a simple average calculation on test cases that has too many covered methods, where the covered methods with lower probabilities have the majority can decrease the value of the test case. In addition, even if the test case covers methods with higher probability, their value will be undermined with average (mean) calculation. Thereby, we have used the lower quartile value of the number of covered methods for each test case, which is inspired by the "*Law of the Minimum (LoM)*." This allowed us to evaluate the test cases fairly. This methodology has prioritized the test cases and used the change impact analysis effectively. Even though the change impact analysis data were used effectively in prioritizing test cases, similar test cases were prioritized consecutively, which sometimes led to a decrease in APFD results. This problem has been discussed in our research question **RQ2**.

RQ2: Does using change impact analysis information solve the problem of similar test cases to be ordered consecutively in test case prioritization? In our research question **RQ1**, we have discussed the usage of change impact analysis in test case prioritization, which is related to research question **RQ2**, we wanted to observe and evaluate if using change impact analysis data will solve the problem in similar test cases to be ordered consecutively or not. We were able to solve the problem of how to change impact analysis data. To solve this problem we differentiated similar test cases by multiplying the ratio of the covered methods, for each test case. In order to increase the priority of such test cases, we used Jaccard similarity. This way, we calculated the number of similarities among test cases and reduced their score with respect to the amount of similarity. We called this test case prioritization method *Dis-LoM-Score*, which performed better than *LoM-Score* in 4 projects and ranked 1st in three of our case study projects *commons-csv*, *jackson-core*, and *jsoup*. The reason why *Dis-LoM-Score* did not perform better *LoM-Score* in the other six projects is because of the coverage criteria granularity we used for calculating similarity. For similarity, we used method coverage, and we found that there were test cases that covered the exact same methods. In other words, there were test cases that had a 100% similarity which annihilated the test score (test score multiplied by 0) that was calculated. Once a 100% similarity between test cases is calculated. The test case score is updated to 0, which will end as a low-priority test case, and will not be given the chance for exposing faults. Therefore, a fine-level granularity (such as statement-level or branch-level) similarity could reduce the possibility of test score annihilation.

C. RUNTIME EVALUATION

In this section, we evaluate the runtime performances of our proposed TCP methods with the other TCP methods we have compared. In Table 10, we present the mean runtime values in milliseconds (ms). We see that the *Dis-LoM-Score* TCP method is the slowest among all other TCP methods. The main reason *Dis-LoM-Score* is slowest is because it uses Jaccard similarity to update test scores after each selection of test case, which is already an expensive process. Even though *Dis-LoM-Score* is the slowest, on average it still performs under 2 minutes, which is still a reasonable time.

We can also see that the *diff*-based TCP methods also have high runtimes compared to the traditional coverage-based TCP method. This is an expected runtime result since these TCP methods have a *diff* process in between.

It is also important to mention that, we haven't included the runtime results of the change impact analysis step for *LoM-Score* and *Dis-LoM-Score*, since that we provide the change impact analysis results separately. However, previous studies [38], [39] that used Code-Change-Sniffer on the Defects4J data have reported that the change impact analysis ran between 15-185 seconds depending on the size of the project. We had consistent runtime results on the change impact analysis from previous studies, which are again within a reasonable runtime.

TABLE 10. Mean of runtime (ms) for every 10 projects from Defects4J and for 6 TCP techniques.

TCP	Total	Additional	Total-Diff	Additional-Diff	LoM-Score	Dis-LoM-Score
commons-csv	1.76	1.76	71.98	74.18	4.62	41.5
commons-codec	3.82	4.16	268.34	274.46	4.62	83.28
gson	8.38	12.84	581.54	563.22	397.66	1514
jackson-databind	15.16	27.04	1158.42	1138.74	17279.7	40117.18
jsoup	2.1	2.52	117.88	120.76	37.66	160.7
jackson-xml	3.88	3.8	191.68	189.28	126.62	276.2
commons-Compress	2.02	2.04	216.88	219.4	9.04	39.52
joda-time	95.26	202.02	1091.26	1141.12	12906.42	100475.82
commons-lang	11	11.96	1044.42	1069.14	42.12	482.78
jackson-core	4.94	5.52	345.08	353.84	179.46	362.2

We also do a complexity analysis on both *LoM-Score* and *Dis-LoM-Score* test case prioritization algorithms. The *LoM-Score* approach iterates the list of test cases to find and select the test case with the next maximum test score, which is actually a sorting operation, that orders test cases in descending order. Thereby, if there are n number of test cases in the list, the complexity of the *LoM-Score* test case prioritization algorithm is $O(n^2)$. The *Dis-LoM-Score* test case prioritization algorithm is quite similar to the *LoM-Score* test case prioritization algorithm. The difference is, before the sorting operation, we first calculate the similarities between test cases using Jaccard similarity, which has a $O(n^2)$ complexity. However, this operation is done only once. Then we select the test case with the highest test score at every iteration, which is a $O(n^2)$ complexity as well. After the highest test score is selected, we update the test score for all the remaining test cases, which takes another n iteration. Therefore, the complexity of the *Dis-LoM-Score* test case prioritization algorithm is $O(n^3)$. To improve the performance, the test score updating operation is implemented with multi-threads and runs in parallel. While updating the test scores, we do not require any synchronization because the updated test score are not dependent on each other. Briefly, we can conclude that the *LoM-Score* test case prioritization algorithm has lower complexity compared to *Dis-LoM-Score* test case prioritization algorithm.

D. THREATS TO VALIDITY

In this section, we discuss the limitations of our overall evaluation that involves our external and internal threats to validity.

1) INTERNAL THREATS TO VALIDITY

While calculating the similarity between test cases, we followed a naive approach by checking if test cases covered the same methods (not statements or branches of a statement). However, the main threat of following this approach is that it will not take into account if the test cases are covering completely different statements or branches of the same method. This type of scenario will prove the opposite of the similar result of method-based coverage. For instance, assume there are two test cases t_1 and t_2, and they both cover the methods m_1, m_2, and m_3. Based on a method-level coverage similarity analysis, the test cases t_1 and t_2 will be calculated as 100% similar. So, if test case t_2 is selected, while we use the *Dis-LoM-Score* TCP approach, the updated test score for t_1, will be 0 and will be set as the lowest priority score. However, in reality, multiple test cases can cover the exact methods, but they could be covering different parts of the methods. Therefore, a finer granularity of coverage similarity would give more accurate results compared to large granularity coverage such as method coverage. On the other hand, using a finer granularity like statements or branches will lead to higher complexity due to the increased number of elements that are covered. The Jaccard similarity is already expensive and would make the *Dis-LoM* TCP even slower. Therefore, there is a trade-off using a finer or larger granularity of coverage criteria. However, our code has been designed in a way to support different coverage criteria, the developer only needs to provide the coverage data.

Another limitation is that this study has not taken into account flaky tests, where test cases have a test execution dependency. Dependent test cases can affect the order of test cases. For instance, a test case might have to run before another test case, due to its setup and initialization that will be used by other test cases. If this test case is not executed before this can lead the dependent test cases to result in failure, which could be a false alarm. Thereby, each TCP approach that we have evaluated, including our proposed TCP methods might have generated infeasible test cases execution order, which can cause flaky tests [40]. Therefore, in future work, we plan to find the dependent test cases and discard the possibility of generating infeasible TCP orders.

Our proposed two TCP methods (*LoM-Score* and *Dis-LoM*-Score) rely on a change impact analysis tool called Code-Change-Sniffer [5]. We used Code-Change-Sniffer because it provides probabilistic results of a method being impacted by a change, which allows us to calculate a score for each test case. Best to our knowledge, other change impact analysis approaches do not provide such information or they do not share their code publicly so that we can use or integrate it into our approach. In addition, Code-Change-Sniffer was reported to have high recall values (very few to no false-negatives), but contain false-positive results. Relying on a change impact analysis that has false-positive results can disorient and mislead the test case prioritization approach and test score calculation. However, if the change impact analysis is replaced with another that has higher recall and precision results, it could improve the test case prioritization outcome.

2) EXTERNAL THREATS TO VALIDITY

We have performed our case study on ten different open-source Java projects, which were obtained and used from Defects4J. However, this case study is not sufficient to generalize results for every Java-based software or other that are developed in different programming languages. Therefore, the selected projects from Defects4J [24] may not represent all the software that is actively been used.

In our case study, the APFD metric is selected for comparing TCP techniques. Moreover, the APFD is an accepted and widely used metric, although in literature the APFD metric is mostly discussed as not taking any consideration of the cost of test case runtimes. For instance, it is possible that two different test execution orders can have the same APFD values. However, at the point, where both of them have detected all the faults could result in different test execution time. Briefly, even if both test execution orders have the same APFD values, they might have different run times.

All the faults in the case study are injected with Major and their representatives of real faults may be argued. There are also studies [24] that provide real faults. However, the faults were all isolated and contained in different commits. Therefore, we couldn't include multiple faults in one project, otherwise, the APFD measure would be meaningful on with a single fault. We could have used the projects that were created by Paterson et al. [25], which includes multiple faults in a project. However, those projects were meant for coverage-based TCP approaches, and not for change-based TCP approaches. For change-based TCP approaches, a base project is also required so that a *diff* could be calculated. Thereby, we had to hand-seed the mutants that were generated by Major. It is important to mention that mutation faults can be a representative of real faults [30], [31]. These studies indicate that it is usually appropriate to use mutation faults for studying regression testing techniques.

While hand-seeding the mutants manually, we tried to include as various types of mutants as possible and tried to include as much as possible that was generated by Major. However, since we have manually selected the mutants and hand-seeded the mutants to the projects, there is still a possibility of this approach being biased. There could be better ways of selecting better mutants that represent real faults or could have increased the number of mutants that are seeded. But again, we have carefully selected the mutation testing tool that is well-known for generating mutants that could represent real faults, such as *Statement-Deletion*.

VII. RELATED WORK

Mirarab et al. [16] proposed an approach to prioritize test cases in the aspect of regression testing to enhance the rate of fault detection. They proposed a unified model based on the probability that uses Bayesian Networks (BN). Their proposed model utilizes data on source code changes, software fault proneness, and test coverage. The approach was compared with nine different prioritization approaches by using APFD (Average Percentage Faults Detected) metric results. They observed that BN has produced better results when the software system contains more faults. Later, Mirarab and Tahvildari [41] extended the approach by adding a feedback route to update the Bayesian Network as prioritization progresses. For instance, if a test case covers a set of program elements, the probability of selecting other test cases that cover the same elements will be lowered. In addition, Ufuktepe et al. [18] extended the work of [16] by providing a fully automated test case prioritization and execution architecture by using different tools. All these studies have used Bayesian Network and focused on the class-level test case prioritization, by using change information, but they do not include information on data dependency. In our study, we have focused on method-level test case prioritization and included data dependency information combined with the flow of method-calling relationships. This information has been used with Markov chains to obtain the reasoning information with respect to the data dependency, change information, and method calling relationships.

Zhao et al. [42] has also used the Bayesian Network (BN) based test case prioritization technique [16], however, they highlighted that the BN-based test case prioritization technique ignores the similarities between test cases that share the same coverage information. Therefore, they have combined the BN-based test case prioritization approach by using clustering for grouping similar test cases. This allows giving higher priority to test cases that are not similar to the test cases that have a higher priority. In other words, the same components will not be covered consecutively with respect to the prioritized test cases. Our study differs by the way we use Markov chains and Jaccard similarity instead of using clustering. Since this study has used the TCP that Mirarab et. al [16] proposed, it follows a class-level test case prioritization approach.

Gupta et al. [43] proposed a regression testing approach by using program slicing. In their study, they have used dataflow-based regression testing that uses two types of program slicing techniques; backward and forward walk slicing techniques. With their technique and using program slicing they intended to determine the directly and indirectly affected define-use (def-use) pairs. With their approach, they reduced the time of maintaining and updating the test suite. Using slicing techniques to find the data dependency relationships is a common method to use, our study is not only limited to using program slicing but also uses change information to strengthen the prioritization outcome.

Jeffrey et al. [44] mentioned the former techniques on test case prioritization that were based on the total number of coverage requirements exercised by the test cases. They presented a new approach to prioritize test cases that considers the coverage requirements present in the relevant slices of the outputs of test cases. In addition, they have implemented three different heuristics based on their relevant slicing-based approach to prioritize test cases and perform their case studies to compare the effectiveness of their techniques with traditional techniques that only account for the total requirement coverage. Their case study and results have shown that using relevant slices for TCP has achieved high rate of fault detection.

Panda et al. [45] proposed a static-based approach to prioritize test cases in regression testing, by computing the affected component coupling of the affected parts of the object-oriented programs. In order to represent the affected parts of the program, they have constructed a graph called the affected slice graph (ASG). The ASG graph is used for determining the fault-proneness of the nodes in the affected slice graph, by computing their corresponding affected component coupling. They prioritize their test cases with respect to the test cases that cover the nodes with higher affected component coupling values.

Wang et al. [46] targeted the test case prioritization problem on service-oriented workflow applications, by highlighting the architecture service-oriented applications that require precise prioritization to execute test cases earlier to detect failures. They have proposed a modification impact analysis-based test case prioritization technique that investigates the internal structure of software, and the fault propagation behavior of modifications.

Tahat et al. [47] proposed and evaluated two test case prioritization methods which are selective methods and dependence-based methods, that utilized the state-based model of the system under test (SUT). Their methods assume that the changes/modifications are made both on the SUT and its model. Furthermore, they have also presented an analytical framework for evaluating test case prioritization methods, in order to reduce the cost of evaluation and comparison. They have performed empirical studies and compared their results with different test case prioritization methods. The results of their empirical studies have shown that system models might improve the effectiveness of test case prioritization, in terms of detecting the faults earlier.

Elbaum et al. [48] highlighted the unawareness of engineers of the relationship between change patterns and testing techniques' cost-effectiveness which can lead to making poor choices in defining regression testing. They have defined the possible three poor choices: (1) designing expensive regression test suites; (2) integration of unnecessarily expensive changes into the system build; (3) inappropriate selection of change integration strategies or regression testing techniques. Therefore, they have performed an empirical study on four different software with several releases and the results have shown that, change attributes play a significant role in the performance of regression testing techniques.

Hao et al. [49] proposed a unified test case prioritization approach that is motivated by the existing additional and total coverage-based test case prioritization techniques. Their approach initially assigns each program element a default probability value denoting the probability that the element contains a fault. When a test case is executed, there is a chance that the test case might have some unit. Therefore, the test case may reveal one or more faults in this unit, and the probability that the unit missed the undetected faults is thus reduced. They have represented the degree of reduction with a ratio between 0.0-1.0 and use this ratio in their unified TCP approach in order to encapsulate the aspects of the additional strategy into the total strategy.

In another study, Emam et al. [50] proposed a fault-based test case prioritization technique for model-based testing procedures. They have proposed an extended directed graph model, which is realized with reinforcement learning and Hidden Markov Model in order to prioritize test cases. Furthermore, they have proposed another alternative test case prioritization approach based on the number of changes that are made in the software. They have applied their test case prioritization approaches to GUI-based applications.

Hemmati et al. [51] proposed a study on prioritizing manual test cases in rapid release environment, by recalling that most of the existing test case prioritization techniques are code coverage based, which actually require access to the source code. However, manual testing is mainly performed in a black-box manner, where the source code is not provided to the testers. Therefore, Hemmati et al. [51] examined existing diversity-based and history-based test case prioritization techniques and modified them to make the test case prioritization technique applicable to the manual black-box system testing. In their study, they have performed empirical studies on several releases of desktop, mobile, and tablet Firefox projects. They have concluded that test cases in rapid-release environments can be very effectively prioritized for execution, based on their historical failure knowledge.

Shin et al. [52] proposed a test case prioritization technique that combines two approaches; mutation-based and diversity-aware. Their technique relies on the diversity-aware mutation adequacy criterion, which was previously proposed by Shin et al. [53], [54]. The diversity-aware criterion's goal is to distinguish the behavior of every mutant from that of all the others, despite the mutants from the original program. Based on the diversity criterion distinguishing the mutants improves the fault detection capabilities in terms of mutation testing. Thereby, their proposed test case prioritization technique gives higher priority to those test cases that help distinguish all mutants as soon as possible. Other studies have also found that mutation-based test cases prioritization [55], [56] to be effective. Since mutation score is used as an indicator of good quality test cases, using it in test case prioritization gives higher priority to test cases that have a higher chance of exposing faults.

Do et al. [57] designed and performed an experiment on Java programs that are tested under JUnit framework. They have found that test case prioritization improves the rate of fault detection for JUnit test suites. In addition, they have also revealed that in terms of test case prioritization, differences can occur with respect to the used programming language and testing paradigm. Later, Do et al. [6], associated the traditional test case prioritization techniques with cost and depending on the testing processes that are employed. On the other hand, Mei et al. [58] proposed a static approach for prioritizing JUnit test cases. They have proposed a TCP approach called JUPTA, which runs on a system with JUnit test cases that functions in the absence of dynamic coverage data.

Nardo et al. [59] presented a case study on coverage-based regression testing techniques on a real-world industrial system with real regression faults given. They have evaluated test case prioritization, selection, minimization, and hybrid approaches that combine selection and minimization techniques. Their study on test case prioritization has shown that techniques that are based on additional coverage with finer-grained coverage criteria perform better in terms of fault detection rates. Furthermore, they have observed that using modification information in prioritization techniques does not significantly enhance fault detection rates.

Eghbali and Tahvildari [60] discussed that acting randomly in the case of ties can degrade the performance of the Additional Technique (AT) algorithm and they have empirically shown that it is very likely for AT to face ties. To break the ties, unlike AT-type techniques which only consider the not-yet-covered entities for coverage, entities are assigned with priorities for further coverage. They used lexicographical ordering to break the ties. In addition, they proposed GeTLO algorithm by modifying and enhancing the basic algorithm in order to reduce its time complexity. However, they showed even with their GeTLO algorithm they have faced ties in test cases. Thereby, they have presented an algorithm that takes into account all possible test cases to find the best test case. Furthermore, they have studied different granularity levels of coverage and they saw that their proposed algorithm has outperformed many of the coverage-based techniques.

Most of the TCP techniques used a dynamic or static analysis approach. Saha et al. [61] followed a different approach to tackle the problem as an information retrieval problem, rather than using traditional dynamic or static analysis approaches. They have used change impact analysis to extract the changes, and they have used the TF.IDF information retrieval model to rank the test cases. The idea of this study intersects with our approach by using change impact analysis information. However, there are several differences between our work. For instance, the authors have used a different change impact analysis approach and used information retrieval techniques to prioritize test cases, while we use Markov chains for change impact analysis and used our formulation (Law of Minimum inspired approach) to prioritize test cases.

Reinforcement learning has also been actively used in several fields, such as trust mechanisms [62], and software security [63]. In software testing, we also see applications of reinforcement learning in test case prioritization. For instance, Bagherzadeh et al. [64] formalized the test case prioritization in Continuous Integration (CI) environment as a reinforcement learning problem, where changes are more frequent in software. Another study [65], has used reinforcement learning in test case prioritization using XCS classifier systems (XCS), which is a genetic algorithm-based learning system that is used in reinforcement learning.

VIII. CONCLUSION AND FUTURE WORK

In this study, a test case prioritization technique called *LoM-Score*, which is based on change impact analysis has been proposed. The proposed TCP technique has been implemented for prioritizing test cases for regression testing. Then the proposed approach has been enhanced with a dissimilarity-based coordinate analysis that uses Jaccard similarity. This approach is called *Dis-LoM-Score*.

In addition, we have used probabilistic results of a change impact analysis tool called Code-Change-Sniffer [5]. The change impact analysis tool follows a method-level granularity approach, which uses method change information, program slicing, and call graph. Then this information is used in a Markov chain model to calculate the probabilistic results.

The proposed test case prioritization approaches *LoM-Score* and *Dis-LoM-Score* are compared with five other TCP techniques. The results show that *LoM* and *Dis-LoM* have performed more consistent results compared to traditional TCP techniques, and shown reliable results. Our statistical tests have shown that there was no significant difference between the *Total-Diff* TCP, which is also the competitive TCP method. However, we have also observed that in some cases *Total-Diff* TCP has performed under the Random TCP, which makes *Total-Diff* an unreliable TCP method.

Testing a code where it is only changed may not be the best approach since that change can have side effects, and these side effects can be the actual code that needs to be tested. Therefore, combining change impact analysis with test case prioritization can have the advantage of guiding test cases to focus on the codes that are impacted by the change. Test cases that are only focused on the changed parts of the code can easily miss the affected codes.

This study has many potentials for improvement. Therefore, for future work, different change impact analysis tools can be tested on the *LoM-Score* and *Dis-LoM-Score* TCP methods to see if there are any significant differences. Furthermore, the granularity that is used for similarity can be replaced with a finer granularity such as; statement level or branch level. A finer granularity could lead to calculating accurate similarities, and better prioritization.

REFERENCES

[1] A. Kumar, "Development at the speed and scale of Google," in *Proc. QCon San Francisco*, 2010, pp. 1–41.

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

[3] B. Athira and P. Samuel, "Web services regression test case prioritization," in *Proc. Int. Conf. Comput. Inf. Syst. Ind. Manage. Appl. (CISIM)*, Oct. 2010, pp. 438–443.

[4] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[5] E. Ufuktepe and T. Tuglular, "Code change sniffer: Predicting future code changes with Markov chain," in *Proc. IEEE 45th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jul. 2021, pp. 1014–1019.

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

[7] D. Binkley, "The application of program slicing to regression testing," *Inf. Softw. Technol.*, vol. 40, nos. 11–12, pp. 583–594, Dec. 1998.

[8] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf.*, Jul. 2010, pp. 373–382.

[9] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: A control call graph based technique," in *Proc. 12th Asia–Pacific Softw. Eng. Conf. (APSEC)*, 2005, p. 9.

[10] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: An eclipse plug-in for testing and debugging," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2012, pp. 378–381.

[11] D. W. Binkley and K. B. Gallagher, "Program slicing," in *Advances in Computers*, vol. 43, Amsterdam, The Netherlands: Elsevier, 1996, pp. 1–50.

[12] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.* Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[13] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[14] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Softw. Test., Verification Rel.*, vol. 23, no. 8, pp. 613–646, Dec. 2013.

[15] R. S. Arnold and S. A. Bohner, "Impact analysis-towards a framework for comparison," in *Proc. Conf. Softw. Maintenance*, 1993, pp. 292–301.

[16] S. Mirarab and L. Tahvildari, "A prioritization approach for software test cases based on Bayesian networks," in *Proc. Int. Conf. Fundam. Approaches to Softw. Eng.* Cham, Switzerland: Springer, 2007, pp. 276–290.

[17] E. Ufuktepe and T. Tuglular, "A program slicing-based Bayesian network model for change impact analysis," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2018, pp. 490–499.

[18] E. Ufuktepe and T. Tuglular, "Automation architecture for Bayesian network based test case prioritization and execution," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, Jun. 2016, pp. 52–57.

[19] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *ACM SIGSOFT Softw. Eng. Notes*, vol. 9, no. 3, pp. 177–184, May 1984.

[20] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge, "A framework for optimizing Java using attributes," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.* Indianapolis, Indiana: IBM Press, 2000, p. 8.

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. CASCON 1st Decade High Impact Papers*. Armonk, NY, USA: IBM Corporation, 2010, pp. 214–224.

[22] G. Jayaraman, V. P. Ranganath, and J. Hatcliff, "Kaveri: Delivering the Indus Java program slicer to eclipse," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.* Cham, Switzerland: Springer, 2005, pp. 269–272.

[23] V. P. Ranganath and J. Hatcliff, "Slicing concurrent Java programs using Indus and Kaveri," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, nos. 5–6, pp. 489–504, Oct. 2007.

[24] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 437–440.

[25] D. Paterson, G. Kapfhammer, G. Fraser, and P. McMinn, "Using controlled numbers of real faults and mutants to empirically evaluate coverage-based test case prioritization," in *Proc. IEEE/ACM 13th Int. Workshop Autom. Softw. Test (AST)*, May 2018, pp. 57–63.

[26] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2011, pp. 612–615.

[27] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for Java," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 433–436.

[28] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (demo)," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Jul. 2016, pp. 449–452.

[29] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of PIT," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2017, pp. 430–435.

[30] J. H. Andrews, L. C. Brand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 402–411.

[31] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[32] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[33] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A mutation system for Java," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 827–830.

[34] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Softw. Test., Verification Rel.*, vol. 15, no. 2, pp. 97–133, 2005.

[35] 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 *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 654–665.

[36] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2000, pp. 102–112.

[37] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Jun. 2002.

[38] E. Ufuktepe, T. Tuglular, and K. Palaniappan, "The relation between bug fix change patterns and change impact analysis," in *Proc. IEEE 21st Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Dec. 2021, pp. 1089–1099.

[39] E. Ufuktepe, T. Tuglular, and K. Palaniappan, "Tracking code bug fix ripple effects based on change patterns using Markov chain models," *IEEE Trans. Rel.*, vol. 71, no. 2, pp. 1141–1156, Jun. 2022.

[40] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 298–311.

[41] S. Mirarab and L. Tahvildari, "An empirical study on Bayesian network-based approach for test case prioritization," in *Proc. Int. Conf. Softw. Test., Verification, Validation*, Apr. 2008, pp. 278–287.

[42] X. Zhao, Z. Wang, X. Fan, and Z. Wang, "A clustering-Bayesian network based approach for test case prioritization," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, vol. 3, Jul. 2015, pp. 542–547.

[43] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proc. Conf. Softw. Maintenance*, 1992, pp. 299–308.

[44] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, 2006, pp. 411–420.

[45] S. Panda, D. Munjal, and D. P. Mohapatra, "A slice-based change impact analysis for regression test case prioritization of object-oriented programs," *Adv. Softw. Eng.*, vol. 2016, pp. 1–20, May 2016.

[46] H. Wang, J. Xing, Q. Yang, D. Han, and X. Zhang, "Modification impact analysis based test case prioritization for regression testing of service-oriented workflow applications," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, vol. 2, Jul. 2015, pp. 288–297.

[47] L. Tahat, B. Korel, M. Harman, and H. Ural, "Regression test suite prioritization using system models," *Softw. Test., Verification Rel.*, vol. 22, no. 7, pp. 481–506, Nov. 2012.

[48] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri, "Understanding the effects of changes on the cost-effectiveness of regression testing techniques," *Softw. Test., Verification Rel.*, vol. 13, no. 2, pp. 65–83, 2003.

[49] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Trans. Softw. Eng. Methodology (TOSEM)*, vol. 24, no. 2, p. 10, 2014.

[50] S. S. Emam and J. Miller, "Test case prioritization using extended digraphs," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 1–41, Dec. 2015.

[51] H. Hemmati, Z. Fang, M. V. Mäntylä, and B. Adams, "Prioritizing manual test cases in rapid release environments," *Softw. Test., Verification Rel.*, vol. 27, no. 6, p. e1609, Sep. 2017.

[52] D. Shin, S. Yoo, M. Papadakis, and D.-H. Bae, "Empirical evaluation of mutation-based test case prioritization techniques," *Softw. Test., Verification Rel.*, vol. 29, nos. 1–2, p. e1695, Jan. 2019.

[53] D. Shin, S. Yoo, and D. Bae, "Diversity-aware mutation adequacy criterion for improving fault detection capability," in *Proc. IEEE 9th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Apr. 2016, pp. 122–131.

[54] D. Shin, S. Yoo, and D. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 914–931, Oct. 2018.

[55] L. Gonzalez-Hernandez, B. Lindström, J. Offutt, S. F. Andler, P. Potena, and M. Bohlin, "Using mutant stubbornness to create minimal and prioritized test sets," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2018, pp. 446–457.

[56] E. Ufuktepe, D. K. Ufuktepe, and K. Karabulut, "MuKEA-TCP: A mutant kill-based local search augmented evolutionary algorithm approach for test case prioritization," in *Proc. IEEE 45th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jul. 2021, pp. 962–967.

[57] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, 2004, pp. 113–124.

[58] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1258–1275, Nov. 2012.

[59] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system," *Softw. Test., Verification Rel.*, vol. 25, no. 4, pp. 371–396, Jun. 2015.

[60] S. Eghbali and L. Tahvildari, "Test case prioritization using lexicographical ordering," *IEEE Trans. Softw. Eng.*, vol. 42, no. 12, pp. 1178–1195, Dec. 2016.

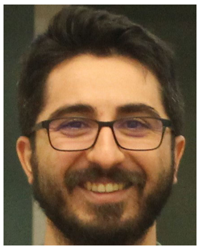
[61] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 268–279.

[62] L. Xiao, Y. Ding, D. Jiang, J. Huang, D. Wang, J. Li, and H. Vincent Poor, "A reinforcement learning and blockchain-based trust mechanism for edge networks," *IEEE Trans. Commun.*, vol. 68, no. 9, pp. 5460–5470, Sep. 2020.

[63] T. Shi, L. Xiao, and K. Wu, "Reinforcement learning based test case prioritization for enhancing the security of software," in *Proc. IEEE 7th Int. Conf. Data Sci. Adv. Anal. (DSAA)*, Oct. 2020, pp. 663–672.

[64] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 2836–2856, Aug. 2022.

[65] L. Rosenbauer, A. Stein, R. Maier, D. Pätzel, and J. Hähner, "XCS as a reinforcement learning approach to automatic test case prioritization," in *Proc. Genetic Evol. Comput. Conf. Companion*, Jul. 2020, pp. 1798–1806.



EKINCAN UFUKTEPE (Member, IEEE) received the B.S. degree in computer engineering from the İzmir University of Economics, in 2011, and the M.S. and Ph.D. degrees in computer engineering from the İzmir Institute of Technology, in 2014 and 2019, respectively. In 2013, he was a Research Intern with the Security and Trust Department, SAP Laboratories France. From 2019 to 2021, he was a Postdoctoral Researcher with the Computational Imaging and Visualization Analysis (CIVA) Laboratory, University of Missouri, Columbia, where he has been an Assistant Professor with the Electrical Engineering and Computer Science Department, since 2022. His current research interests include software testing, program analysis, and software security.

TUGKAN TUGLULAR (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Ege University, Turkey, in 1993, 1995, and 1999, respectively. He was a Research Associate with Purdue University, from 1996 to 1998. He has been with the İzmir Institute of Technology, since 2000. After that, he became an Assistant Professor with the İzmir Institute of Technology, where he was the Chief Information Officer, from 2003 to 2007. Currently, in addition to his academic duties, he acts as an IT Advisor to the Rector. He has more than 60 publications and an active record of duties with international and national conferences. His current research interests include model-based testing and model-based software development.