EFFECTIVENESS OF USING CLUSTERING FOR TEST CASE PRIORITIZATION

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Can GÜNEL

July 2019
İZMİR

We approve the thesis of **Can GÜNEL**

Examining Committee Members:

Assoc. Prof. Dr. Ahmet Tuncay ERCAN
Department of Management Information Systems, Yaşar University

Assoc. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering, İzmir Institute of Technology

Assoc. Prof. Dr. Tolga AYAV
Department of Computer Engineering, İzmir Institute of Technology

25 July 2019

Assoc. Prof. Dr. Tolga AYAV
Supervisor, Department of Computer Engineering
İzmir Institute of Technology

Assoc. Prof. Dr. Tolga AYAV
Head of the Department of
Computer Engineering

Prof. Dr. Aysun SOFUOĞLU
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my supervisor, Assoc. Prof. Dr. Tolga Ayav, who guided me since the beginning of my education, and encouraged me during this thesis study. I am so grateful to him for his patience and guidance. It was a big pleasure for me to work with him.

In addition, I would like to express my infinite gratitude to my family for their unconditional love, and endless support. It is the most perfect feeling in the world to know that my family is always there from the toughest to the most cheerful moments.

ABSTRACT

EFFECTIVENESS OF USING CLUSTERING FOR TEST CASE PRIORITIZATION

Software testing is one of the most important processes in the software development life cycle. As software evolves, previous test cases need to be re-executed to make sure that there is no new bugs introduced and nothing is broken in the existing behaviours. However, re-execution of all test cases could be expensive. That is why, test case prioritization method can be used to detect faults earlier by prioritizing the test cases which could have the higher possibility than others to find faults.

Studying different approaches, implementing different techniques or putting these techniques to test on different programs could make it easier to answer which technique should be used for which kind of programs or faults. We address this issue, focusing on selecting different test case prioritization approaches and calculating the average fault detection ratios of prioritized test suites. As a novelty, we propose to perform an optimization algorithm on one of the approaches called ‘Clustering‘ to increase its efficiency. To do that, our main objective is determined as maximizing the distance between each clusters by using the coverage information. The distance is measured as the difference of covered functions of test cases in a test suite. In the end, this study will give a hint about selection of test case prioritization technique to be used by checking the empirical results of the experiments.

ÖZET

TEST DURUM ÖNCELİKLENDİRMESİNDE KÜMELEME KULLANIMININ ETKİLİLİĞİ

Yazılım testi, yazılım geliştirme döngüsünün en önemli süreçlerinden birisidir. Çünkü, yazılım büyüdükçe, yeni hatalar ortaya çıkarılmadığından ve çalışan hiçbir fonksiyonun bozulmadığından emin olmak için önceden tanımlanmış testlerin tekrar çalıştırılması gerekmektedir. Ancak bu testlerin tekrar çalıştırılması işlemi oldukça maliyetli olabilir. Bu sebeple, yazılımdaki hataları daha erken tespit edebilmek amacıyla, hatayı bulma olasılığı daha fazla olan testleri önceliklendirmeye dayanan test durum önceliklendirmesi metodu kullanılabilir.

Çeşitli yöntemler üzerinde çalışarak, çeşitli teknikleri geliştirerek ve bunları çeşitli programlar üzerinde test ederek, hangi tekniğin hangi tip programlarda yada hangi tip hatalarda kullanılabileceğine daha kolay cevap verilebilir. Biz bu konuda test durum önceliklemeye yöntemlerinin seçimi ve önceliklendirilmiş test grubunun ortalama hata bulma oranlarını hesaplanmasına odaklanarak gerçekleştirdik çalışmamızı. Yenilik olarak son zamanlarda kullanılmaya başlanan yöntemlerden biri olan Kümeleme metoduna verimi artımak için optimizasyon uygulanmasını öneriyoruz. Bunu yapabilmek için, esas olarak kümeler arasındaki mesafenin kapsam bilgileri kullanılarak maksimum olmasını amaçlıyoruz. Kümeler arasındaki mesafe kümelerin kapsadığı fonksiyonlar arasındaki fark ile hesaplanmaktadır. Sonuç olarak, bu çalışma deneysel sonuçlara bakarak hangi test durum önceliklendirme yönteminin seçilebileceği konusunda ipucu vermektedir.

To my precious family

Contents

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

LIST OF ABBREVIATIONS

TCP . . . . . Test Case Prioritization
TS . . . . . Test Suite
RT . . . . . Random Testing
ART . . . . . Adaptive Random Testing
SA . . . . . Simulated Annealing
APFD . . . . . Average Percentage of Faults Detected
GT . . . . . Greedy Total
GA . . . . . Greedy Additional
JD . . . . . Jaccard Distance
CMD . . . . . Coverage Manhattan Distance
HD . . . . . Hamming Distance
RQ . . . . . Research Question
SIR . . . . . Software-artifact Infrastructure Repository

Chapter 1

INTRODUCTION

1.1. Thesis' Aim and Objectives

A software development is a long-termed modularised process and generally more than one developers participate it. Thus, at any step when a change is made by anyone, all test cases introduced before need to be re-executed [1]. This process is called regression testing. In order to reveal faults as quick as possible, test case prioritization (TCP) can be applied to regression test suite(TS) [2, 3].

TCP is one of the hottest research topic in software testing [1, 3, 4, 5, 6] and there are various TCP techniques in the literature. Therefore, we think of optimizing a rarely used technique, improving its efficiency and verifying its results by comparing with the frequently used techniques.

While we have this idea in our mind, we started to search most frequently used TCP techniques along with an optimizable TCP technique. Random testing(RT) is the most basic testing approach [7, 8]. There are some TCP techniques which are derived from RT. One of them is called Adaptive Random Testing (ART) prioritization which uses test case similarity based on the coverage information [9]. Another common approach for TCP is Greedy. This approach is also based on the coverage information of the test cases. During our search, we noticed that in recent years Clustering approach is started to be used for TCP. Thus, we decided to optimize Clustering approach. To ease the comparison between approaches, coverage information is used for all TCP techniques specified in this theses study.

After completing this preparatory work, we started to implement two versions of ART-based prioritization approach and two versions of Greedy approach. Then we developed our proposed optimized Clustering approach by using Simulated Annealing optimization algorithm. As a result we compared all these five techniques with respect to their fault detection ratios.

1.2. Organization of Thesis

The thesis is organized as follows. Chapter 2 covers all the general information regarding software testing, including the analysis of different TCP techniques. Chapter 3, points out our proposal for optimizing TCP techniques. Chapter 4 gives the objective and requirements of the proposed technique and then provides the experimental works, together with the results. We conclude the thesis in Chapter 5 and provide a future work.

Chapter 2

TEST CASE PRIORITIZATION

2.1. Introduction

Software testing is a very important phase of software development. Software testing is a quality control process of evaluating a system or its components. This process aims to verify and validate the product by checking whether the actual results match the expected results. The testing process comprises executing a program with the purpose of finding possible bugs in the system. Thus, software testing should be taken place during the development to ensure that the product is bug free before it is released to end user. In this context, the analysis of software testing should consider some important research questions listed below:

- Why is software testing important?
- What are the software testing levels?
- What kind of software testing methods exist?
- What is software testing constraints?
- What are the techniques used to make testing process effective?

To find answer of these questions, first, we need to analyze the background of software testing. Hence, the main purpose of this chapter is to comprehend the concept of software testing in literature.

The rest of this chapter is organized as follows. In Section 2.2, fundamentals of software testing are presented. This sections also shows how and why software testing is important in software development process. Furthermore, information regarding software testing levels, software testing methods and regression testing and its techniques are provided. In Section 2.2.4.3 test case prioritization subject is introduced and some existing methods and related works are explained.

2.2. Fundamentals of Software Testing

Software testing is a necessary process of software development in order to discover defects in the software and to make sure the software satisfies the requirements of end users. Developers make use of this process by fixing the defects discovered in this process and deliver a good quality product. Thus, in order to deliver a good quality product, software testing process should be conducted very carefully. To conduct an effective testing process, well planned test cases and scenarios need to be prepared. A test case or a test scenario is one of the number of steps to be followed in the testing process. A test case includes determining the input values, expected outputs, preconditions and post-conditions. A test case describes how a function or module needs to be tested according to determined inputs, outputs and conditions. A set of test cases constitute a test suite. Thus a test suite contains all conditions of a software to be tested. A suitable test suite should cover as much as possible parts of the software.

2.2.1. Importance of Software Testing

A defect or an error in software may cause a system failure. This failure can led to monetary or even human loss. There are many examples where software bugs have led to loss of life or millions of dollars in losses. Some of them are listed below:

- The Mariner 1 Spacecraft [10, 11] worth of 18 millions of dollars was crashed immediately after lift off. Later it is found that a missing hyphen caused wrong guidance signals to be sent to the rocket.

- In mid-December 1989, AT&T installed new software in 114 electronic switching systems. On January 15, 1990, 5 million calls were blocked [12, 13] during a 9 hours period nationwide. The bug was traced to a C program that contained a break statement within a switch clause nested within a loop. Initially, the loop contained only if clauses with break statements to exit the loop. AT&T wound up losing 60 millions of dollars in charges that day.

- Ariane 5 [11, 13, 14] was a European rocket designed to launch commercial payloads such as communications satellites into Earth orbit. The rocket is exploded 37 seconds after take off. The reason of the explosion is an attempt to convert a 64-bit

floating point number representing the horizontal velocity to a signed 16-bit integer caused the number to overflow. More than 370 millions of dollars were lost due to this error.

- Therac-25 [15] was a radiation machine used in cancer treatment. The machine was administered electron beams to treat surface tumours and x-rays to treat deep tumours. Therac-25 was started giving dosages from 75 to 100 times stronger than normal to some patients. 6 patients have died because of overdose radiation. A race condition in the software had caused x-rays to be used instead of electron beams.

- On February 25, 1991, during the Gulf War, a scud missile was launched by Iraq to barracks in Dhahran, Saudi Arabia. There was a Patriot missile defense system in Dhahran and normally the Patriot should have detected the missile and destroyed it. However, the Patriot was detected the missile launched by Iraq in wrong position, almost 600 meters away the actual position. This miscalculation caused 28 U.S. soldiers death and injured 100 others [16]. Later it has been found that a software error in Patriot's system that handling timestamps has led this tragic failure.

As seen in the above examples, the bugs in the software used widely in every part of daily life may cause a critical and bad results. In order to prevent these bugs it should be attached great importance to software testing process. If software testing is not performed properly, software, applications or systems can have errors which may lead to rework, costly failure or worse, loss of life.

2.2.2. Levels of Software Testing

There are four main levels of testing that should be done before a software is released or delivered to end users. These are Unit Testing, Integration Testing, System Testing and Acceptance Testing. Why Regression Testing is not included? Because regression testing is not a separate level; it is just a type of testing that can be performed during any or all of the main testing levels. Figure 2.1 [17] shows the general structure of software testing levels. In the following subsections, these main levels of software testing are explained.

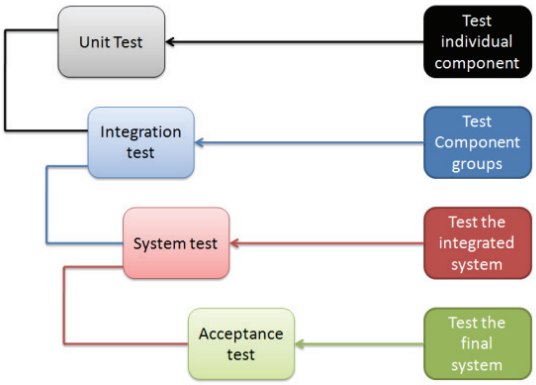

Figure 2.1. Overview of Software Testing Levels

2.2.2.1. Unit Testing

A unit [18] is the smallest, individual and testable part of the source code. The idea behind of unit testing is to divide the program into units and verify each units of the product. This testing is conducted by developers to make sure that their code is working well and meets the user specifications. Unit testing is important for finding software bugs early, facilitating changes needed for bugs, simplifying integration and improving the quality of source code.

2.2.2.2. Integration Testing

Integration testing is a type of testing to check if combination of different pieces of the modules or programs work together as a cluster. The main purpose of this level of testing is to find the bugs during interaction of integrated units of the project. A lot of functional, requirement and performance faults are revealed in this type of testing [17]. In unit testing it is verified that different units work as per the requirement individually but in integration testing it is verified that whether these individual units work as per requirement after they are integrated together. Integration testing is performed by the software test engineers. There are two common approaches of integration testing.

1. **Top down:** In this approach, the upper level units are tested earliest and lower levels are tested step-wise after upper levels [17]. This approach is preferred when

top-down development methodology is used [19]. In case low level units are not available at the beginning of the test execution, in order to simulate lower level units, Test Stubs are needed.

2. **Bottom up:** This approach is just opposite of the Top down approach. In this approach, the lower level units are tested earliest and upper levels are tested step-wise after lower levels [17]. This approach is preferred when bottom-up development methodology is used [19]. In case high level units are not available at the beginning of the test execution, in order to simulate higher level units, Test Drivers are needed.

2.2.2.3. System Testing

System testing is a type of testing to verify all well integrated units of the product as a whole. In this type of testing, the software is tested with all aspects including hardware compliance, performance and adherence to quality standards [17]. System testing is performed by skilled test team. In order to test the system effectively, testers need to create scenarios similar to the ones in real world where the system will be deployed.

2.2.2.4. Acceptance Testing

Acceptance testing is the final testing phase of software before it is delivered to end user or customer. In this type of testing, the software is tested for correctness and completeness [17]. Acceptance testing is formal testing performed by end user of the software and/or quality assurance team with respect to many aspects such as overall functionality, cosmetic looking, user requirements or business contract compliance etc.

2.2.3. Methods of Software Testing

Test cases are written using different kinds of techniques in order to make software testing process more efficient [20]. With these techniques, testers can increase the fault detection rate. The methodologies that includes these techniques can be considered as the set of testing mechanisms used in software development life-cycle. Deciding a suitable

testing methodology is considered to be the core of the testing process. There are two major methodologies for a software to be tested:

1. Black-box Testing
2. White-box Testing

In the following subsections these methodologies are explained.

2.2.3.1. Black-box Testing

Black-box testing is based on deriving tests from external descriptions of the software, including specifications, requirements and design. This type of testing looks at things from the end user's perspective. Tester does not have knowledge of internal program design and source code [20] as seen in Figure 2.2 [21]. The tester has only a set of input values and corresponding expected results [22]. On giving input, if the output equals to the expected results, the test result is "OK", it is "PROBLEMATIC" otherwise. Black-box testing is performed by test team generally. Acceptance testing is an example of block-box testing.

Figure 2.2. Black-box Testing

2.2.3.2. White-box Testing

White-box testing is based on deriving tests from the source code internals of the software, specifically including branches, individual conditions and statements. This type of testing requires knowledge of internal program design and source code [20] as seen in Figure 2.2 [21]. Each test case is an attempt to analyze the logic of the source code [22].

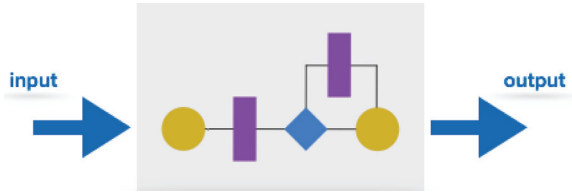

Figure 2.3. White-box Testing

White-box testing is performed by developers generally. Unit testing is an example of white-box testing.

Code Coverage analysis is most widely used technique for white-box testing. Code Coverage analysis points out the gaps in a test suite. It specifies the statements, branches or functions of the program that are not visited by the test suite. As soon as the gap is specified according to the code coverage analysis, test cases should be written for the non-visited/non-tested parts.

2.2.4. Regression Testing

Regression testing is the process of verifying of any code change or modification on a software does not cause to unexpected results. It simply confirms or denies software's functionality. Code changes are always done on a software because of fixing existing bugs, adding new features to the software or customizing the software according to the end user's needs etc. When a code change is done, it is essential to make sure that the software's older functionalities still work with new changes. It should be guaranteed that by fixing a bug or changing even a small thing have not broken another thing. This can be achieved by doing the regression testing. Regression testing can be performed using any combination of the following techniques; Retest All, Test Case Selection and Test Case Prioritization. In the following sections, these techniques are discussed and detailed information regarding our main topic -test case prioritization- are given.

2.2.4.1. Retest All

As the name implies, the main objective of this technique is that all of the test cases in a test suite should be re-executed. This technique may result in the execution of unnecessary test cases. Thus, retest all technique is very expensive since it requires great amount of time and resources.

2.2.4.2. Test Case Selection

In this technique, a representative subset of test cases are selected from a test suite. Selection should be done considering the modifications that have been made on the software. Thus, test case selection technique can be divided into two parts. First one is identifying the affected parts of the software and second one is extracting related test cases from test suite. By applying this technique, only selected test cases are executed. This could decrease the retesting effort and cost. However, selection should be done very carefully because there is a possibility to selected test cases miss the potential faults in the changed software.

2.2.4.3. Test Case Prioritization

The main goal of this technique is to prioritize the test cases in a test suite in a such manner that more critical and effective test cases execute before than others. Test case prioritization is simply ordering test cases in a test suite according to their ability to reveal faults and effectiveness. Prioritization should depend on the impact of the test cases and frequently used functionalities.

2.3. Analysis of Test Case Prioritization Techniques

Regression testing is a very important part of software development. Because as software grows, risks of making mistakes increases. In order to minimize these risks and to make sure that added parts do not break anything, the software should be tested

periodically. However, when software gets bigger in size, it gets more costly to execute [23]. Moreover, there could be limited resources or time constraints for test execution. Thus, a technique or a process is needed which makes the testing process more efficiently.

As briefly explained in section 2.2.4, there are various methods for regression testing. One of these methods is called test case prioritization which could be used to make regression testing more efficiently by revealing faults as early as possible. In this method, test cases are reordered by prioritizing more important and useful test cases according to chosen goal. This goal can be either maximizing code coverage or minimizing the human participation or maximizing average percentage of faults detected (APFD) etc. This could help to reduce usage of resources and to handle time constraints. Below is the general definition of TCP [2]:

Definition Let T a test suite, PT the set of permutations of T, and f a function from PT to the real numbers. The goal is to find $T' \in PT$ such that $(\forall T")\ (T" \in PT)\ (T" \neq T')\ [f(T') \geq f(T")]$

In the definition, PT stands for all possible orderings of T and f denotes a function applied on PT. f is also called as *award value*. Higher award values are favoured to lower ones.

2.3.1. Existing TCP Methods

In literature, there are various techniques to be used to prioritize test cases. These techniques are divided into two groups as Static and Dynamic TCP [6]. In this study, we focused on dynamic TCP techniques. In this section, we introduced several existing and widely used dynamic TCP techniques to be later used on comparing our proposed method.

2.3.1.1. Adaptive Random Testing

In this TCP technique, random selection of test cases is the main idea. The first step of this technique is selection of a set of test cases randomly from non-prioritized test cases to generate a candidate set. Then a test case is chosen from the candidate set

which is most distant from the prioritized set of test cases. In order to determine the most distant test case, the code coverage-based pair-wise distance of test cases is calculated. There are 3 different distance functions proposed as maxmin, maxavg, maxmax. A test case is chosen that has the largest minimum distance according to maxmin function, that has the largest average distance according to maxavg function and that has the largest maximum distance according to maxmax function with the prioritized set. The test case to be selected according to the distance function should increase the coverage. Figure 2.4 is the visualization of the steps of this ART technique.

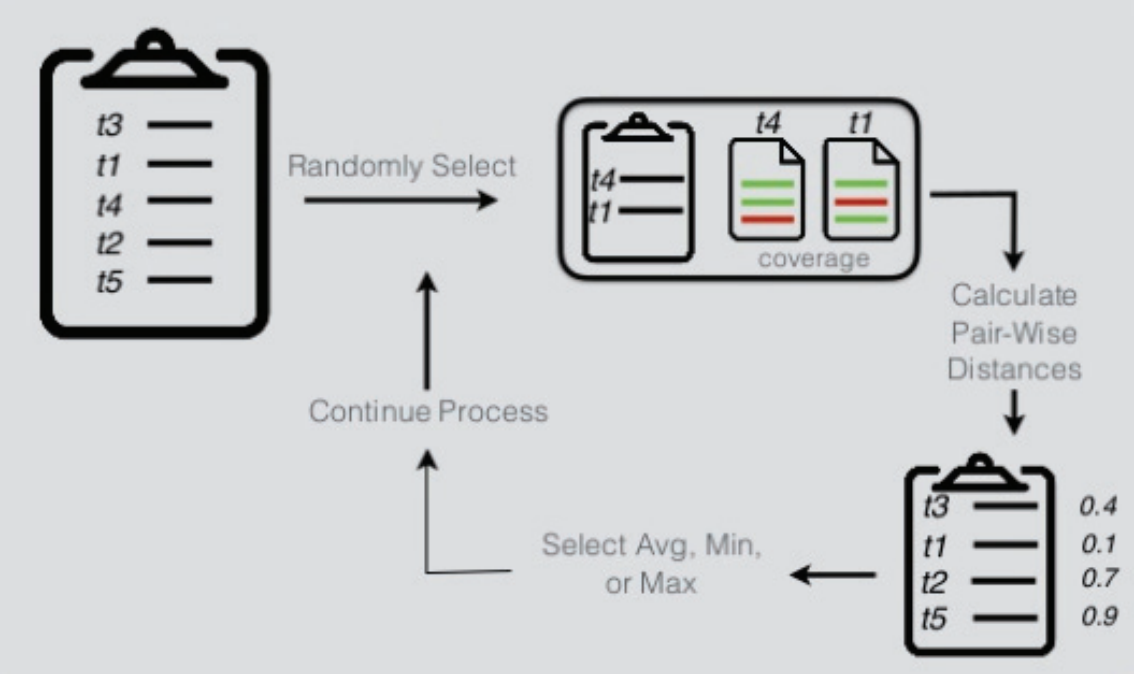

Figure 2.4. The process of ART Technique

In our study, we implemented two types of this technique. The first one is proposed by Jiang et al. The second one is proposed by Zhou et al. with some small differences from the one proposed by Jiang et al. The details and differences are explained further in the section 2.3.2.

2.3.1.2. Greedy Approach

Traditionally, dynamic TCP methods use two policies, the total policy and additional policy, to prioritize test cases based on the code coverage. The total policy prior-

itizes test cases based on their code coverage, and the additional policy prioritizes test cases based on their code coverage excluding the code elements that have been covered by prioritized test cases. Thus, the greedy total(GT) policy favors the test cases that cover more code, but the greedy additional(GA) policy would select the test cases that can cover different code from the already prioritized test cases earlier. The figure 2.5 explains better the differences between these two policies. The additional policy of greedy approach has been commonly accepted as one of most effective TCPs in previous works [24, 25]. Recently, Zhang et al. proposed a novel approach to bridge the gaps between these two strategies by unifying the strategies based on the fault detection probability [25, 26].

Statement	Test case 1	Test case 2	Test case 3
1	X	X	X
2	X	X	X
3		X	X
4		X	
5			X
6			X
7	X		X
8	X		X
9	X		X

Total statement coverage: **(TC3, TC1, TC2)**
Additional statement coverage: **(TC3, TC2, TC1)**

Figure 2.5. The Differences Between Two Greedy Policies

In our study, we implemented both of the greedy policies based on the work by Rothermel et al. [27] and also mentioned work in the section 2.3.2.

2.3.2. Related Works

Jiang et al. [24] proposed an ART-based prioritization technique which uses a dynamic candidate set in order to prioritize test cases. The steps of ART technique are

presented in section 2.3.1.1. In the pair-wise distance calculation step, Jaccard distance (JD) technique is used in this study to measure the distance between the selected candidate test case and the prioritized set of test cases and test selection is done according to the maxmin distance function. The steps mentioned in section 2.3.1.1 are repeated until there are no more non-prioritized test cases.

Zhou et al. [28] proposed an ART-based prioritization technique similar to the one proposed by Jiang et al [24]. Zhou uses a fixed size (i.e., 20) candidate set instead of creating it dynamically and in the step of choosing a most distant test case from candidate set, he uses the Coverage Manhattan distance (CMD) technique while JD is used in [24]. The rest of the process is same with [24].

Rothermel et al. [2] used greedy approach in his study to find an optimal prioritization. Greedy approach is based on selecting the test cases that reveals most faults which are not yet revealed by the previously selected test cases until test cases that reveals all faults selected. Another version of the greedy approach is used for coverage. In this version, test cases that covers most code or statement or branch or function are selected to prioritize the test cases in a test suite.

Pang et al. [29] proposed a clustering approach for prioritizing test cases. In this study, test cases are split into two groups(clusters) as effective and non-effective through K-Means clustering algorithm. It applies Hamming distance(HD) on coverage information of present and previous versions of the program under test in order to calculate the differences. According to this differences effective and non-effective clusters are created. After creating clusters, the test cases in effective clusters have higher priorities and executed on a modified version of program under test.

Carlson et al. [30] proposed a hierarchical clustering approach for TCP. They split the test cases into groups according to their code coverage similarities. After creating clusters, the test cases in a cluster are ordered according to their code coverage, code complexity and fault history information. In order to obtain a final prioritized set of test cases round robin fashion is used which means picking a test case from each cluster respectively until all test cases are exhausted.

Chapter 3

PROPOSED TCP TECHNIQUE

3.1. Introduction

In this chapter, we present our proposed TCP technique, including a description of the implemented procedures. Furthermore, we provide a running example of the proposed technique so that the whole procedure can be demonstrated in detail.

3.2. Overview of the Proposed Technique

Proposed technique for test case prioritization process contains mixture of some of the used methods in existing TCP techniques such as test case similarities and grouping test cases. Our new method is simply based on clustering approach which is used in the recent years. An optimization algorithm (Simulated Annealing) is embedded to the new method as a part of TCP process.

To sum up, the proposed method is optimizing the clustering approach according to the test case similarity. Similarities between test cases are calculated by using the coverage information of the test cases. Section 3.3 describes the proposed algorithm which can be used prioritizing test cases in a test suite in detail. In the algorithm simulated annealing optimization method is used as a part of prioritization phase in order to get a better results in our clustering approach. To have a better insight, the algorithm is split into two procedures.

3.3. Procedures

First of all, in our algorithm, a data structure is needed to represent a non-prioritized test suite. As shown in OPTIMIZED CLUSTERING procedure, a dictionary (map) is kept

to represent the test suite according to the given input file which contains test cases and respective coverage information. Then the test suite is divided into 5 groups (clusters) randomly and as even number of test cases in each cluster as possible. After clusters are created, SA optimization takes place to classify clusters as good as possible. The Pseudo-code of our SA optimization algorithm is shown in APPLY SA OPTIMIZATION procedure [31].

1: **procedure** OPTIMIZED CLUSTERING($inputFile$) ▷ $inputFile$ is a file containing coverage information of a sample program
2: $TestSuite \leftarrow inputFile.load$ ▷ a test suite is a dictionary where each key is a test case id and each value is corresponding coverage information of a test case as covered function, branch or statement ids
3: $Clusters \leftarrow randomPartition(TestSuite, NumberOfClusters)$
4: $Clusters \leftarrow applySAOptimization(Clusters)$
5: $Clusters \leftarrow sortEachCluster(Clusters)$
6: $PrioritizedTestSuite \leftarrow pickTestCases(Clusters)$ ▷ a prioritized test suite is a list of test case ids
7: **end procedure**

Before moving on our customized SA algorithm, first a high-level overview of SA is explained in this paragraph. The simulated annealing algorithm was originally inspired from the process of annealing in metal work [32]. Annealing involves heating and cooling a material to alter its physical properties [33]. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties. In simulated annealing, a temperature variable is kept to simulate this heating process. It is initially set to high value and then it is slowly cooled as the algorithm runs. While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optimums. As the temperature is reduced so is the chance of accepting worse solutions [34]. Below is the general steps of this algorithm:

1. Set initial temperature and generate a random initial solution.
2. Calculate the initial solution's cost using some cost function.
3. Generate a random neighboring solution by making a small change to our current solution (aka. perturbation function).
4. Calculate the new solution's cost.

5. Compare them:

   If new cost $<$ old cost: move to the new solution, saving it as the base for next iteration.

   If new cost $>$ old cost: *maybe* move to the new solution. Most of the time, the algorithm will avoid moving to a worse solution. If it did that all of the time, though, it would get caught at local maxima. To avoid that problem, it sometimes elects to keep the worse solution. To decide, the algorithm calculates something called the 'acceptance probability' and then compares it to a random number.

6. Decrease the temperature.

7. Repeat steps 3-6 above until an acceptable solution is found or the system has sufficiently cooled or some maximum number of iterations is reached.

In our study, SA optimization's main purpose is to maximize the distance between clusters in each iteration. To calculate the distance between clusters, each test cases in each clusters are compared with each other according to their coverage information and the number of differences on coverage between two test cases give the distance value of these two test cases. Each test cases are compared iteratively and get a total distance of two clusters by adding the all distances calculated. These steps are repeated for all possible cluster pairs and all calculated distances are added up to get a final total distance, in other words objective function. The goal is to maximize this final total distance. If the final total distance is bigger than previous iteration, then this distance is accepted. If not, according to SA optimization there is still a possibility to accept it. If this probability is not satisfied either, then the distance is rejected. At each iteration, there are two possible perturbations defined in the proposed algorithm as below;

1. Exchanging two random test cases from two random clusters. (Applied even number of iterations)

2. Moving a random test case from a random cluster to another random cluster. (Applied odd number of iterations)

After applying one of the perturbations, next iteration is processed until reaching the number of maximum iterations. In our study 200 iterations is used as maximum. When all iterations are exhausted, there are five clusters as distinct from each other as possible. In order to generate a prioritized set of test cases, round robin fashion is used which picking a test case from each cluster respectively.

```
procedure APPLY SA OPTIMIZATION(inputFile)
    Find a random initial solution s := s_0
    Select an initial temperature t := t_0 > 0
    Select a temperature reduction function a
    repeat
        repeat
            s' := perturbationFunction(s)
            s' := calculateDistanceForNewClusters(s')
            δ := F(s') − F(s)
            if (δ <= 0) or (exp(−δ/t) < rand[0, 1]) then
                s := s'
            end if
        until maximum distance calculation count is reached
        t := CoolingSchedule(t)
    until maximum cycle count is reached
end procedure
```

3.4. A Running Example of the Proposed Technique

In this section, a test suite example is prioritized by our proposed method step by step. Initially, our technique requires an input file which contains test cases and coverage information. Figure 3.1 shows an example input file that our implementation needed. In this figure, each line represents a test case and each space separated numbers in a line represents an ID of function covered by that test case.

gzip-function.txt

```
11 13 15 16 19 34 72 73
11 13 15 16 19 34 72 73
11 13 17 19 34 72 73
11 13 17 19 34 72 73
11 13 18 19 34 72 73
11 13 18 19 34 72 73
1 2 3 4 6 7 8 9 10 11 13 19 21 23 25 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 72 73 80 81
1 2 3 4 6 7 8 9 11 13 19 20 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 72 73 80 81
11 13 19 20 27 34 36 37 38 41 42 43 64 65 66 67 69 70 72 73
10 11 13 19 21 22 23 24 25 26 27 29 30 31 32 34 36 37 38 41 42 43 64 65 66 67 69 70 71 72 73
11 13 19 20 27 34 36 37 38 41 42 43 64 65 66 67 69 70 72 73
11 13 19 20 27 34 36 37 38 41 42 43 64 65 66 67 69 70 72 73
1 2 3 4 6 7 8 9 10 11 13 19 21 22 23 24 25 26 29 30 31 32 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 71 72 73 80 81
1 2 3 4 6 7 8 9 11 13 19 20 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 72 73 80 81
1 2 3 4 6 7 8 9 10 11 13 19 21 22 23 24 25 26 29 30 31 32 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 71 72 73 80 81
1 2 3 4 6 7 8 9 11 13 19 20 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 72 73 80 81
1 2 3 4 6 7 8 9 11 13 19 21 22 23 24 25 26 29 30 31 32 33 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 71 72 73 80 81
1 2 3 4 6 7 8 9 10 11 13 19 21 22 23 24 25 26 29 30 31 32 33 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 71 72 73 80 81
1 2 3 4 6 7 8 9 11 13 19 21 22 23 24 25 26 29 30 31 32 33 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 71 72 73 80 81
11 13 19 20 27 34 36 37 38 41 42 43 64 65 66 67 69 72 73
10 11 13 19 21 23 25 27 34 36 37 38 41 42 43 64 65 66 67 69 72 73
11 13 19 20 27 34 36 37 38 41 42 43 64 65 66 67 69 72 73
1 2 3 4 6 7 8 9 10 11 13 19 21 22 23 24 25 26 29 30 31 32 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 71 72 73 78 80 81
1 2 3 4 6 7 8 9 11 13 19 20 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 72 73 78 80 81
1 2 3 4 6 7 8 9 10 11 13 19 21 22 23 24 25 26 29 30 31 32 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 71 72 73 80 81
```

Figure 3.1. Example Input File

At this point, our algorithm has all the information and input needed. Thus, the procedures explained in section 3.3 can be started to execute.

Step 1 - Loading Test Suite: First step is loading the input file in a key-value

manner. The keys are test case IDs (TC1, TC2 etc.) and the values are list of function IDs covered by the corresponding test cases. According to the input file given in figure 3.1, there are 25 test cases in our example test suite. Figure 3.2 shows the keys of loaded test suite.

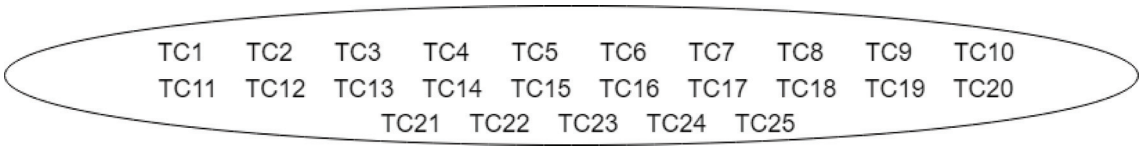

Figure 3.2. Initial Test Suite

Step 2 - Creation of Clusters: Test cases are prorated to 5 Clusters randomly. Figure 3.3 represents the initial status of the Clusters after the test cases are distributed.

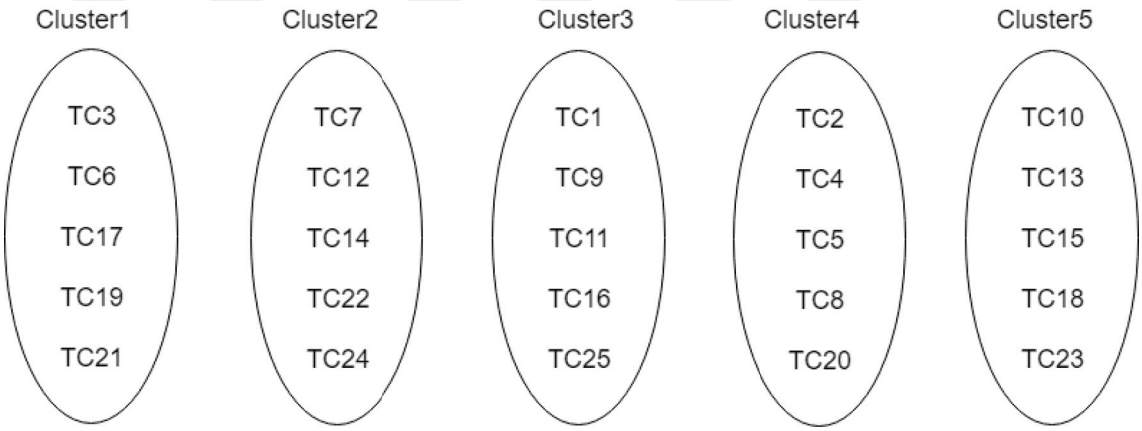

Figure 3.3. Initial Clusters

Step 3 - Calculating The Distance Between Clusters: Prioritization process is started with this step by trying to maximize the distance between clusters. The distance between each cluster pairs are calculated against the coverage differences of the test cases. Below is an example of calculating distance between Cluster1 and Cluster2 step by step.

- Cluster1 and Cluster2 have 5 test cases TC3, TC6, TC17, TC19, TC21 and TC7, TC12, TC14, TC22, TC24 respectively.
- It can be observed the covered function IDs by the test cases mentioned above from the figure 3.1.

- Compare covered function IDs of each test case pairs as shown in figure 3.4. The number of differences of covered function IDs between two test cases give the distance value of compared test cases. For example; the distance between TC3 from Cluster1 and TC7 from Cluster2 is equal to the number different IDs that exist in TC3 and TC7 which is 32 as shown in figure 3.5 (The IDs with red underline are common in both test cases, that is why they are not count). After calculating the distances between each test cases, a total distance value between Cluster1 and Cluster2 is obtained by adding each distance value together.

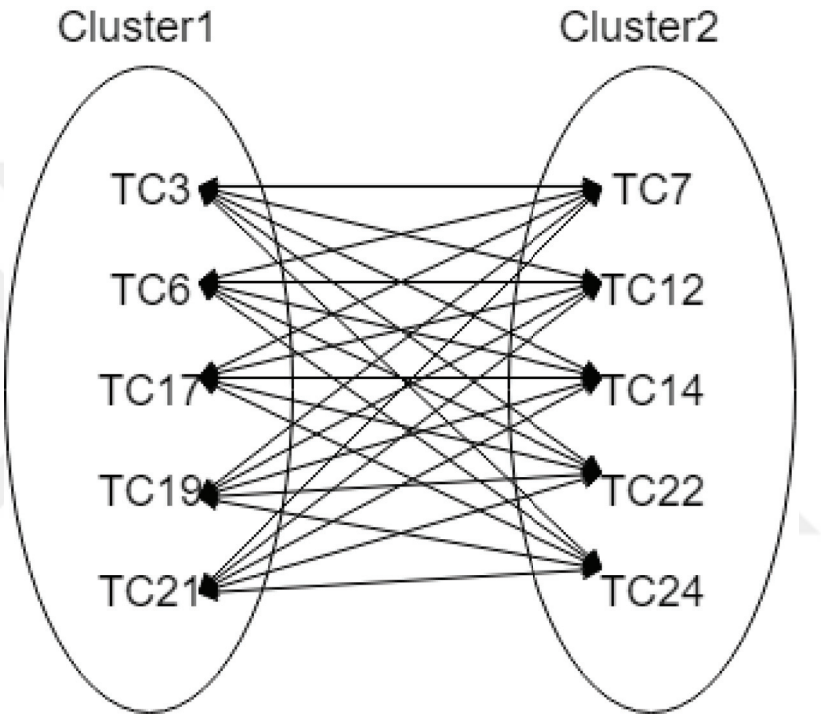

Figure 3.4. Test Case Comparisons

```
11 13 17 19 34 72 73

1 2 3 4 6 7 8 9 10 11 13 19 21 23 25 34 45 46 47 48 49 50 51 52 53 54 55 56 57 65 66 68 70 72 73 80 81

...
```

Figure 3.5. The difference between TC3 and TC7

- Above given steps for calculating the distance between Cluster1 and Cluster2 are repeated for each Cluster pairs.

Step 4 - Calculating Total Distance: After calculating the distance of each Cluster pairs, a total distance value is calculated by adding each specific distance values of each Cluster pairs. For our example, there are 10 cluster pairs distance values as shown in figure 3.6. The sum of these 10 distance values give the total distance value of all Clusters.

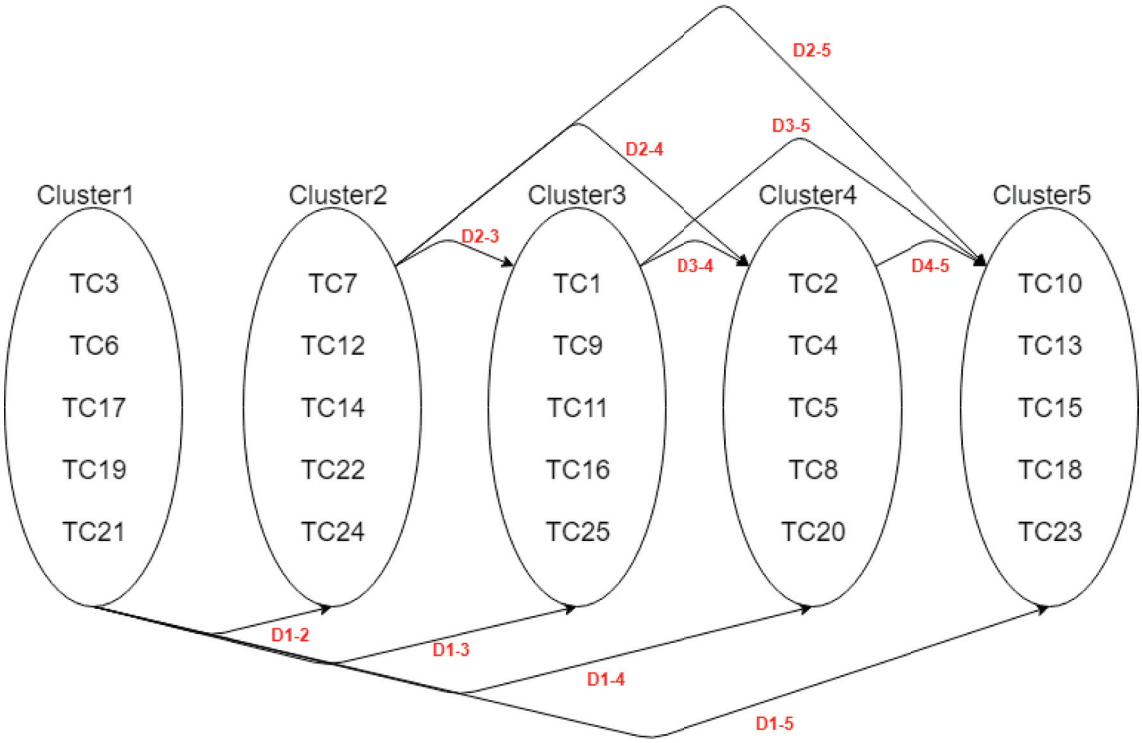

Figure 3.6. Distance Calculation

Step 5 - Applying Perturbations to Maximize the Total Distance: At each iteration, before calculating the total distance as mentioned above in steps 3 and 4, perturbations are applied. In our study there are two perturbation functions. At even number of iterations, perturbation function would be exchanging random test cases from random clusters. At odd number of iterations, perturbation function would be moving a random test case from a random cluster to another random cluster. Figure 3.7 shows an application of perturbation function based on exchanging random test cases between Cluster 3 and Cluster 4.

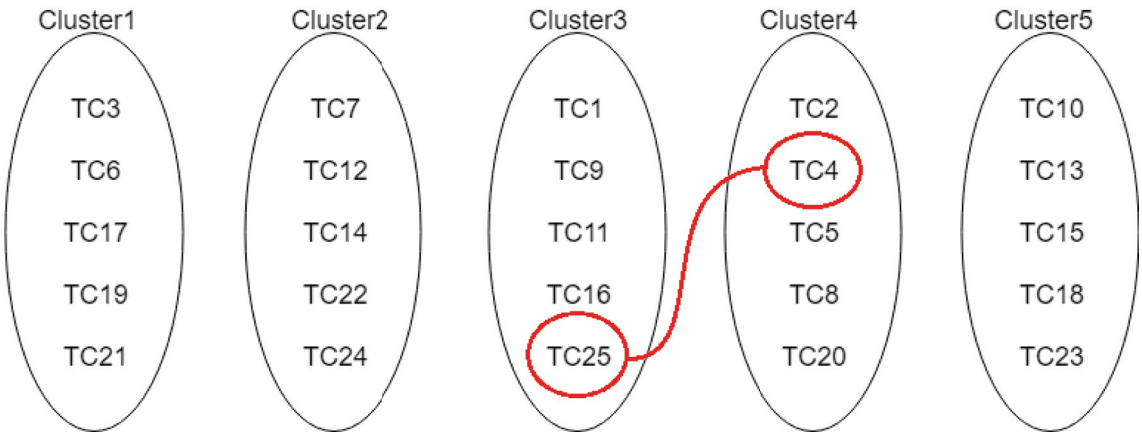

Figure 3.7. Exchanging random test cases from random clusters

Step 6 - Creating Prioritized Test Suite: As a last step, when all iterations are exhausted, clusters are put into their final forms as shown in figure 3.8. (In our example, final forms of the clusters have equal number of test cases but it may not be equal for other situations.) When all clusters are in their final state, a prioritized test suite can be created by picking test cases from clusters in round robin fashion. In our case, the resulted prioritized list would be as; TC7 - TC2 - TC1 - TC14 - TC10 - TC8 - TC4 - TC3 - TC16 - TC13 - TC18 - TC6 - TC5 - TC17- TC15- TC23 - TC9 - TC12 - TC20 - TC19 - TC25 - TC11 - TC21 - TC24 - TC22 respectively.

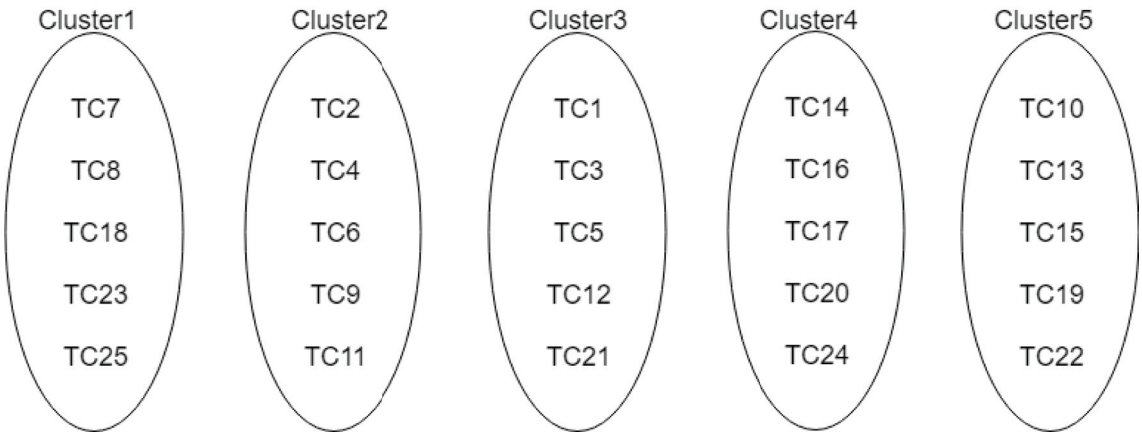

Figure 3.8. Final forms of the clusters after all iterations are exhausted

3.5. Conclusion

This chapter presented the proposed new technique by optimizing the clustering based approaches using test case similarities according to their coverage information. Each execution step of the proposed technique is simulated using a sample test suite.

In the remaining part of this thesis study, we implemented some of the existing TCP techniques along with the one we proposed. Then, we presented a metric called APFD that we use to measure the effectiveness of the implemented prioritization techniques. We calculated the corresponding APFD values of each TCP techniques on different programs. Finally we explained our experiments and compare the APFD results of implemented techniques.

Chapter 4

VERIFICATION OF THE PROPOSED TCP METHOD

4.1. Introduction

This section first introduces the verification metric used to asses the effectiveness of TCP methods, and then continues with the experimental works of this theses study by focusing on the implementation details of the proposed and other existing TCP methods and exhibits the results of the subject programs used in experiments.

In order to measure the effectiveness of a TCP method, its APFD values need to be calculated [2, 1]. This value shows how quickly a prioritized set of test cases reveals faults in the program [1]. Therefore, in this theses study, we focused on this the most important research question (RQ) of TCP, presented below.

RQ. How does OPTIMIZED CLUSTERING algorithm when comparing to other existing algorithms in terms of APFD value?

This research question is related to the APFD results of reordered (prioritized) TS for different TCP techniques. Our aim is to increase this value as much as possible. APFD value is calculated with equation 4.1 [1].

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \tag{4.1}$$

where T is a test suite with n test case, F is a set of m faults detected by T and TF_i is the first test case's position that detects fault i. In section 4.2, APFD metric is explained in detail.

4.2. Average Percentage of Faults Detected Metric

Most of the prioritization techniques concentrate on increasing the rate of fault detection capability of reordered test cases. In order to assess fault detection ratio, APFD

metric is used widely [35]. APFD metric measures the weighted average number of the faults detected during the execution of the test suite. APFD values range from 0 to 100 [36]. Higher APFD value means better or faster fault detection rates. In a plot which shows the number of test cases executed versus the number of faults detected, the area under the curve gives the APFD value as shown in figure 4.1 [37].

Figure 4.1. APFD: The area under the curve

4.2.1. Illustration of APFD Calculation

To illustrate APFD calculation by using the equation 4.1, an example program is considered which contains 10 faults and a test suite with 10 test cases as shown in table 4.1. An intersection point (”X” mark in the table) shows that the fault in the corresponding row is detected by the test case in the corresponding column. With these information, APFD values of prioritized and non-prioritized test suite can be calculated. In order to calculate the APFD value, the parameters in the equations are determined as below:

No. of test cases (n) = 10

No. of faults (m) = 10

The position of the first test in T that exposes fault i. = Tfi

The non-prioritized order according to fi is:

TC1 - TC2 - TC3 - TC4 - TC5 - TC6 - TC7 - TC8 - TC9 - TC10

Note: Figure 4.2 shows that the faults found by non-prioritized test suite.

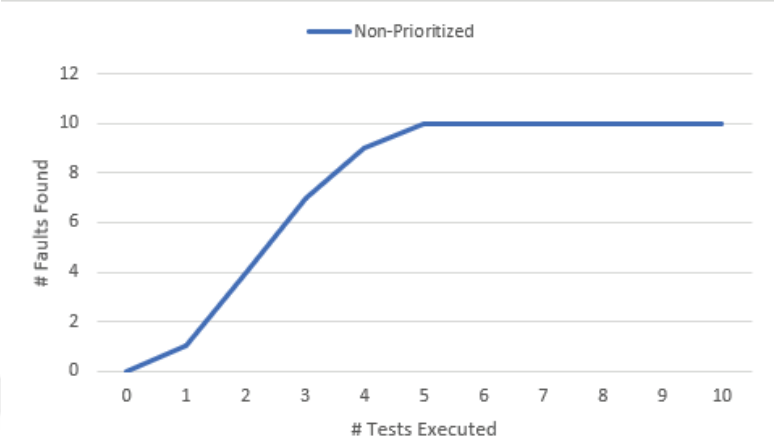

Figure 4.2. Found Faults for Non-Prioritized Test Suite

APFD value for non – prioritized test cases:

APFD = 1 – (2 + 4 + 2 + 3 + 3 + 3 + 5 + 4 + 1 + 2) / (10*10) + 1/(2*10)

= 1 – 29 / 100 + 1 / 20

= 1 – 0.29 + 0.05

= 0.76

Note: Figure 4.3 shows the APFD of the non-prioritized test suite.

Figure 4.3. The area(APFD) Covered by Non-Prioritized Test Suite

The prioritized order according to fi is:

TC3 - TC4 - TC2 - TC1 - TC5 - TC7 - TC6 - TC10 - TC9 - TC8

Note: Figure 4.4 shows that the faults found by prioritized test suite.

Figure 4.4. Found Faults for Prioritized Test Suite

APFD value for prioritized test cases:

APFD = 1 – (1 + 2 + 3 + 1 + 1 + 1 + 5 + 2 + 4 + 3) / (10*10) + 1/(2*10)

= 1 – 23 / 100 + 1 / 20

= 1 – 0.23 + 0.05

= 0.82

Note: Figure 4.5 shows the APFD of the non-prioritized test suite.

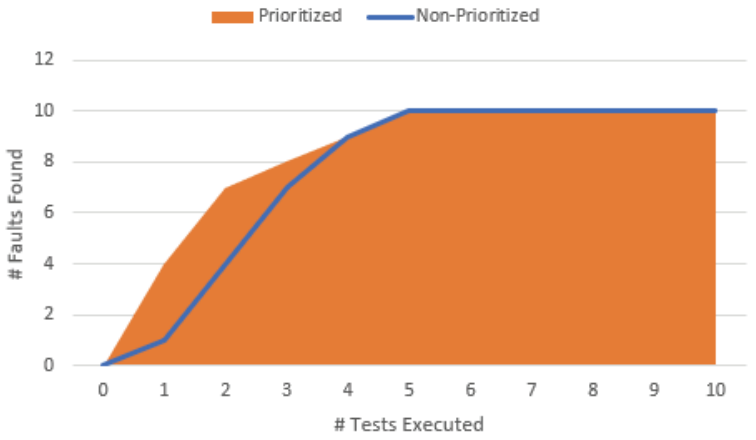

Figure 4.5. The area(APFD) Covered by Prioritized Test Suite

Table 4.1.: Fault Matrix

Faults	**Test Cases**									
	$TC1$	$TC2$	$TC3$	$TC4$	$TC5$	$TC6$	$TC7$	$TC8$	$TC9$	$TC10$
$Fault1$		X	X							
$Fault2$				X	X					
$Fault3$		X				X				
$Fault4$			X						X	
$Fault5$			X	X						
$Fault6$			X				X			
$Fault7$					X		X			
$Fault8$				X						
$Fault9$	X					X			X	
$Fault10$		X		X						

As a result of the APFD calculation in this section, prioritized test suite is detected faults earlier than non-prioritized one. This result is visualized and can be seen in figure 4.5.

4.3. Experimental Work

During implementing any of the TCP methods covered in this theses study, first thing which should be understood is the input file format. This is essential because the test suite is created according to this input file format. Input files are created per subject program. The name of an input file is assumed as <subject>-function.txt. These files contain the coverage information as function ids of each test cases of the subject program. An example can be checked in figure 3.1. The format of the files are as following:

- Each line contains the coverage vector of one test case (line 1 is the coverage of TC#1, line 2 is the coverage of TC#2, and so on).
- The numbers in each line are the IDs of the functions covered by that particular test case.

For verification purposes, 10 subject programs are used in this thesis study. 5 of them are C programs and other 5 are Java programs. Details of these subject programs (i.e. number of lines of the source codes, number of test cases, number of existing faults, fault types etc.) can be found in Table 4.2. C programs are taken from Software-artifact Infras-

tructure Repository (SIR) [38]. Java programs are selected from the Defects4J framework [39]. In order to get the coverage information of these subject programs a linux coverage tool called gcov and SIR tools were used [9]. We have implemented a python script that parses the fault matrix file generated by these tools and creates the input file we used in our experiments.

Table 4.2.: Details of subject programs

$SubjectProgram$	$\#ofLines$	$\#ofTests$	$\#ofFaults$	$FaultType$	$Language$
$flex\_v3$	10296	670	9	$seeded$	C
$grep\_v3$	10124	809	8	$seeded$	C
$gzip\_v1$	4594	214	7	$seeded$	C
$make\_v1$	14330	875	19	$seeded$	C
sed_v6	13413	370	6	$seeded$	C
$closure\_v0$	90697	8124	101	$real$	$Java$
$lang\_v0$	21787	2322	39	$real$	$Java$
$math\_v0$	84323	3877	7	$real$	$Java$
$chart\_v0$	96382	2278	26	$real$	$Java$
$time\_v0$	27801	4160	27	$real$	$Java$

Considering the above information regarding the inputs and subject programs, the process of verification includes following steps:

- Extracting coverage information of the test cases.
- Applying TCP technique.
- Generating a prioritized test suite.
- Calculating the APFD value of the prioritized test suite.

This process is also displayed in figure 4.6.

The experimental work covers the analysis and comparison of 4 different TCP techniques (2 ART-based, 2 Greedy-based) with the proposed optimized clustering-based TCP technique on the subject programs. After applying a TCP technique, APFD value of the prioritized test suite is measured. These steps are repeated 30 times in order to be considered as the normal distribution. This is because multiple test cases may be assigned the same grade by the TCP technique, then random test cases are selected randomly to break the tie and this could change the APFD results.

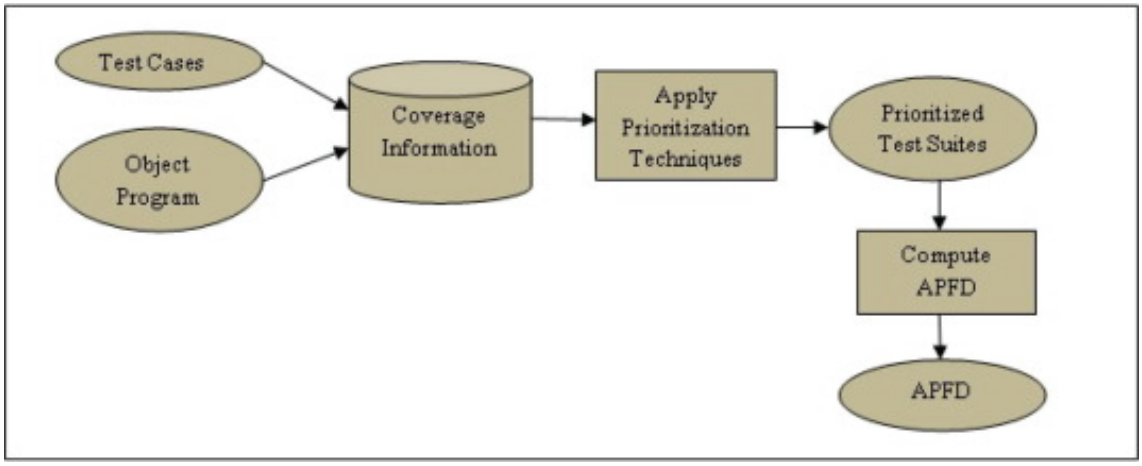

Figure 4.6. Verification Process

4.4. Results and Discussions

This section shows the results of our study and presents an analysis according to the results. Table 4.3 shows the arithmetic average of calculated APFD values for each TCP technique including our proposed one per subject program. The values for the existing TCP methods in the table 4.3 coincides with the results of the study conducted by Miranda et al. [1].

Table 4.3.: Average APFD Results per Subject Program

$SubjectProgram$	$GreedyAdditional$	$GreedyTotal$	$DynamicART$	$FixedART$	$Clustering$
$flex\_v3$	$0,8738$	$0,7703$	$0,9192$	$0,9192$	$0,8441$
$grep\_v3$	$0,8919$	$0,8736$	$0,9639$	$0,9559$	$0,8686$
$gzip\_v1$	$0,8347$	$0,5907$	$0,8558$	$0,7988$	$0,8901$
$make\_v1$	$0,6749$	$0,6778$	$0,6937$	$0,6777$	$0,9218$
sed_v6	$0,9465$	$0,9398$	$0,9519$	$0,9436$	$0,8398$
$closure\_v0$	$0,4993$	$0,5345$	$0,5228$	$0,5275$	$0,4815$
$lang\_v0$	$0,5693$	$0,5052$	$0,4972$	$0,5108$	$0,3694$
$math\_v0$	$0,7036$	$0,5374$	$0,6184$	$0,5982$	$0,3512$
$chart\_v0$	$0,6850$	$0,6103$	$0,5434$	$0,5481$	$0,5770$
$time\_v0$	$0,5252$	$0,5481$	$0,5256$	$0,5054$	$0,5190$

As seen in table 4.3, our proposed method outperforms other existing TCP methods for some subject programs written in C language. Regarding to Java subjects, Greedy approaches performs better than others. In order to analyze the reason behind these results, we investigated the coverage information of each test case per subject program. We implemented a python script that shows the average similarities of test cases of a subject

program. In this script, below steps are followed in order to extract a similarity value of test cases from their respective coverage information.

1. Calculate the similarities of a test case by comparing it with each test case in the test suite.

2. Sum up all the similarity values of the test case to get a total similarity value for that specific test case.

3. Divide the total similarity value to the number of test cases to find the average similarity of the test case.

4. Keep a map where key is the test case ID and value is the corresponding similarity value.

5. Repeat Steps 1-4 for each test case in a test suite to find the average similarity of each test case. Figure 4.7 shows an example map for these calculations.

6. To extract a average similarity value of the subject program, all average similarities of all test cases are added up and divided to the number of test case.

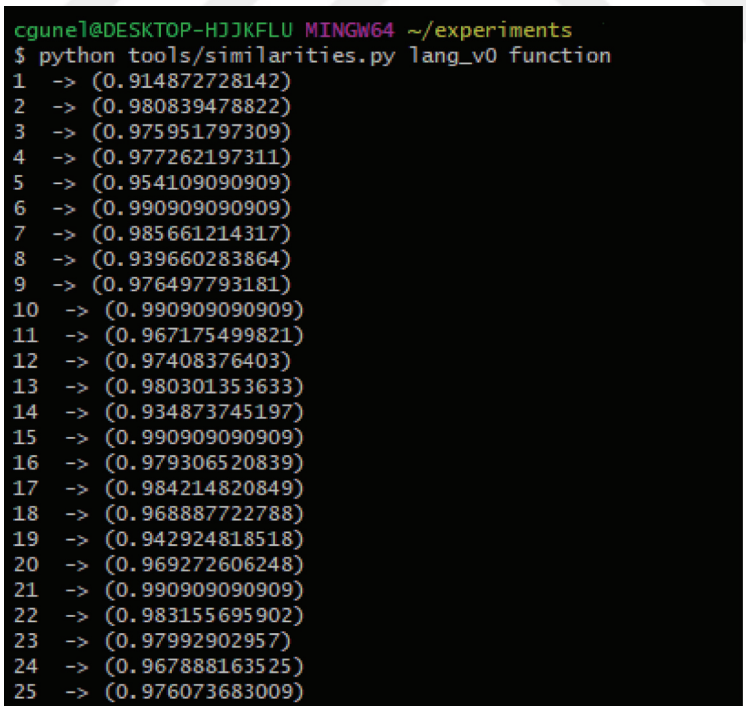

Figure 4.7. Example Map of Avrg. Similarities for Each Test Cases in a Test Suite

The results of the above process per subject program is displayed in table 4.4. According to this table, it can be said that test case similarities of Java programs are much higher than the ones of C programs. This means that, the coverage information of test cases for Java programs is more similar than C programs. Therefore, these results might suggest that our proposed TCP method performs better when the test cases' coverage information similarities are lower. In addition, as seen in table 4.2, the number of tests are highly different between Java and C subjects. Hence, we can come up with a result as high number of tests does not mean high coverage. According to the table 4.2, Java subjects have higher number of tests but according to table 4.4, Java subjects have also higher similarity values. This means that for our Java subjects, test cases mostly covers the same functions. Since this study is focused on TCP based on test case coverage and similarities between test cases, the number of test cases does not affect APFD results.

Table 4.4.: Average Similarity Values of Test Cases per Subject Program

$SubjectProgram$	$AverageSimilarity$
$flex\_v3$	$0,1592$
$grep\_v3$	$0,3281$
$gzip\_v1$	$0,1967$
$make\_v1$	$0,1302$
sed_v6	$0,2649$
$closure\_v0$	$0,6110$
$lang\_v0$	$0,9653$
$math\_v0$	$0,9056$
$chart\_v0$	$0,9401$
$time\_v0$	$0,4479$

4.5. Conclusion

This section covered experimental work for the implementation of the proposed and existing TCP approaches. The experiments mainly focused on the APFD results of the prioritized test suites. To observe the results, experiments was performed on 10 different subject programs, 5 C programs [38] and 5 Java programs [39]. As seen in table 4.3, results showed that the APFD results differ by TCP technique and subject program.

Chapter 5

CONCLUSIONS AND FUTURE WORK

In this thesis study, an analysis of software testing was performed. It has been tried to explain the such questions like why software testing is important, how software testing should be done etc.

For this purpose, the first step was analyzing the background of software testing to comprehend its scope, and the state-of-art methods used in the literature. It was noticed that, in order to perform more efficient testing process, TCP can be applied to test suite. There are various TCP techniques which can be used to handle time and/or resource constraints. Thus, 3 different approach was selected to implement in the scope of this theses study. The first approach was randomized approach. Two different techniques was implemented for this approach which are Dynamic ART and Fixed ART. Second one is greedy approach. Two different techniques was implemented for greedy as well which are greedy total and greedy additional. And last approach is Clustering which contains an optimization process in this study.

The study continued with experimental works and comparing the implemented TCP techniques on different subject programs by using APFD values of the final prioritized test suites as a metric. Results showed that (i) APFD values vary according to the subject program, (ii) APFD values vary according to TCP technique used, and (iii) Clustering approach which includes an optimization process can have better APFD values for some C subjects, however it can be said that this approach is not recommended for programs that have test cases with similar coverage information because of the results in tables 4.3 and 4.4.

To sum up, performing efficient software testing requires understanding the main steps of the TCP. That is why, optimizing the existing TCP techniques or introducing new ones became our goal. With this aim, we applied SA algorithm to Clustering approach in order to increase testing efficiency. In the process of SA algorithm some parameters (i.e. initial temperature, number of iteration etc.) needs to be predefined. Also for grouping similar test cases, the number of clusters needs to be set in the first place. Therefore, a study can be done as a future work to find the optimal values of the predefined parameters.

BIBLIOGRAPHY

[1] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 222–232, New York, NY, USA, 2018. ACM.

[2] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

[3] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[4] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 523–534, May 2016.

[5] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, Sep 2013.

[6] Qi Luo, Kevin Moran, and Denys Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 559–570, New York, NY, USA, 2016. ACM.

[7] Richard Hamlet. *Random Testing*, page 970–978. American Cancer Society, 2002.

[8] PS Loo and WK Tsai. Random testing revisited. *Information and Software Technology*, 30(7):402 – 417, 1988.

[9] Z. Q. Zhou. Using coverage information to guide test case selection in adaptive random testing. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 208–213, July 2010.

[10] Chris Higgins. On this day in 1962, nasa launched and destroyed mariner 1, July 2017. http://mentalfloss.com/article/502943/day-1962-nasa-launched-and-destroyed-mariner-1.

[11] Matt Lake. 11 infamous software bugs, September 2010. https://www.pcworld.com/article/205318/11_infamous_software_bugs.html?page=2.

[12] David Pogue. 5 most embarrassing software bugs in history, November 2014. https://www.scientificamerican.com/article/pogue-5-most-embarrassing-software-bugs-in-history/.

[13] Abhimanyu Grover. Some of software's darkest failures from recent history, 2019. https://testcollab.com/blog/software-darkest-failures/.

[14] Jamie Lynch. The worst computer bugs in history: The ariane 5 disaster, September 2017. https://www.bugsnag.com/blog/bug-day-ariane-5-disaster.

[15] Jamie Lynch. The worst computer bugs in history: Race conditions in therac-25, September 2017. https://www.bugsnag.com/blog/bug-day-race-condition-therac-25.

[16] Michael Barr. Lethal software defects: Patriot missile failure, March 2014. https://embeddedgurus.com/barr-code/2014/03/lethal-software-defects-patriot-missile-failure/.

[17] Test Institute. Software testing level, 2019. https://www.test-institute.org/Software_Testing_Levels.php.

[18] Ekaterina Novoseltseva. 8 benefits of unit testing, January 2017. https://dzone.com/articles/top-8-benefits-of-unit-testing.

[19] Software Testing Fundamentals. Integration testing, 2019. http://softwaretestingfundamentals.com/integration-testing/.

[20] Irena Jovanović. Software testing methods and techniques. *Multi-, Inter-, and Trans-*

disciplinary Issues in Computer Science and Engineering, 5(1):30–41, Jan 2009.

[21] TutorialsPoint. Software testing overview, 2019. https://www.tutorialspoint.com/software_engineering/software_testing_overview.htm.

[22] Ankita Sethi. A review paper on levels, types techniques in software testing. *International Journal of Advanced Research in Computer Science*, 8(7):269–271, Aug 2017.

[23] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74 – 93, 2018.

[24] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, Nov 2009.

[25] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201, Piscataway, NJ, USA, 2013. IEEE Press.

[26] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Trans. Softw. Eng. Methodol.*, 24(2):10:1–10:31, December 2014.

[27] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, Aug 1999.

[28] Z. Q. Zhou, A. Sinaga, and W. Susilo. On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites. In *2012 45th Hawaii International Conference on System Sciences*, pages 5584–5593, Jan 2012.

[29] Y. Pang, X. Xue, and A. S. Namin. Identifying effective test cases through k-means clustering for enhancing regression testing. In *2013 12th International Conference on Machine Learning and Applications*, volume 2, pages 78–83, Dec 2013.

[30] R. Carlson, H. Do, and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 382–391, Sep. 2011.

[31] E. Aycan and T. Ayav. Solving the course scheduling problem using simulated annealing. In *2009 IEEE International Advance Computing Conference*, pages 462–466, March 2009.

[32] M. M. Keikha. Improved simulated annealing using momentum terms. In *2011 Second International Conference on Intelligent Systems, Modelling and Simulation*, pages 44–48, Jan 2011.

[33] G. Jianlan, C. Yuqiang, and H. Xuanzi. Implementation and improvement of simulated annealing algorithm in neural net. In *2010 International Conference on Computational Intelligence and Security*, pages 519–522, Dec 2010.

[34] Nader Azizi and Saeed Zolfaghari. Adaptive temperature control for simulated annealing: A comparative study. *Comput. Oper. Res.*, 31(14):2439–2451, December 2004.

[35] Mr Anil Mor. Evaluate the effectiveness of test suite prioritization techniques using apfd metric. *IOSR Journal of Computer Engineering*, 16:47–51, 01 2014.

[36] Thillaikarasi Muthusamy and Dr. K. Seetharaman. Effectiveness of test case prioritization techniques based on regression testing. *International Journal of Software Engineering Applications*, 5:113–123, 11 2014.

[37] Muhammed Maruf Öztürk. A bat-inspired algorithm for prioritizing test cases. *Vietnam Journal of Computer Science*, 5(1):45–57, Feb 2018.

[38] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled ex-

perimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[39] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.