EFFECTIVE SOFTWARE FAULT LOCALIZATION BASED ON COMPLEX NETWORK THEORY

ABUBAKAR ZAKARI

FACULTY OF COMPUTER SCIENCE AND
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2019

EFFECTIVE SOFTWARE FAULT LOCALIZATION BASED ON COMPLEX NETWORK THEORY

ABUBAKAR ZAKARI

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2019

UNIVERSITY OF MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: Abubakar Zakari

Matric No: WHA160019

Name of Degree: DOCTOR OF PHILOSOPHY

Title of Project Thesis ("this Work"): Effective Software Fault Localization Based On Complex Network Theory

Field of Study: Software Engineering (Program Debugging)

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This Work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date:

Subscribed and solemnly declared before,

Witness's Signature Date:

Name:

Designation:

ABSTRACT

Effective debugging is necessary for producing high quality and reliable software. Fault localization plays a vital role in the debugging process, and it is also the most tedious and expensive activity in program debugging. As such, effective fault localization techniques that can identify the exact location of faults are most needed. Despite various fault localization techniques proposed, their application in multiple-fault programs is limited. The presence of multiple faults in a program reduces the efficacy of existing fault localization techniques to locate faults effectively due to fault interference phenomenon. Moreover, most of these techniques are unable to localize multiple faults simultaneously in a single diagnosis rank list. This has led researchers to adopt approaches such as one-bug-at-a-time debugging approach (OBA) and parallel debugging approach. However, using OBA debugging approach increases software time-to-delivery and potentially leads to more faults during regression testing, while utilizing *k*-mean clustering algorithm with Euclidean distance metric to group failed tests based on their execution profile similarity in parallel debugging approach is claimed to be problematic and inappropriate. This work aims to conduct an investigative study of the claimed problematic parallel debugging approach in comparison with OBA debugging approach and MSeer parallel debugging approach in terms of localization effectiveness. Furthermore, two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), are proposed to improve localization effectiveness, and to aid developers' to localize multiple faults simultaneously in a single diagnosis rank list. The proposed techniques rank faulty statements based on their behavioral abnormalities and distance between program statements in both passed and failed tests execution. In the case where a developer has checked 70% of the program statements and cannot fully localize all the multiple faults in a single diagnosis rank list, rather than resorting to using OBA

debugging approach, a newly proposed community-based fault isolation approach that makes use of a divisive network community clustering algorithm is applied to aid in isolating faults into different fault-focused communities (clusters), each targeting a single fault. In the proposed fault isolation approach, a community weighting and selection mechanism is introduced to aid in prioritizing highly important communities for developers to debug the faults simultaneously in parallel. The experiments performed on several varied multiple-fault programs show that even though the claimed problematic parallel debugging approach is more effective in comparison with the OBA debugging approach, it is not as effective when compared with MSeer parallel debugging approach. The experimental results also show that FLCN-M is much more effective in locating multiple faults in comparison with two similarity coefficient-based techniques (i.e. Ochiai coefficient and Tarantula coefficient) and an existing fault localization technique based on software network centrality measures. FLCN-S shows significant improvement in terms of localization effectiveness on single-fault programs in comparison with Ochiai-based and Jaccard-based techniques. The proposed community-based fault isolation approach shows significant improvement where it performs significantly better in terms of localization effectiveness in comparison with the claimed problematic parallel debugging approach and MSeer parallel debugging approach. Overall, the proposed techniques and approach show improvement in fault localization effectiveness in both single-fault and multiple-fault programs over the baseline techniques and approaches.

Keywords: Complex Network; Software Fault Localization; Program Debugging; Parallel Debugging; Multiple Faults.

ABSTRAK

Penyahpepijatan berkesan adalah perlu bagi menghasilkan perisian yang boleh dipercayai dan berkualiti. *Fault localization* memainkan peranan yang penting dalam proses nyahpepijat, dan ia juga merupakan aktiviti paling membosankan dan mahal dalam penyahpepijatan aturcara. Oleh yang demikian, teknik *fault localization* yang berkesan yang berupaya mengenalpasti lokasi sebenar sesar tersebut adalah amat diperlukan. Walaupun banyak teknik *fault localization* yang dicadangkan, aplikasi mereka dalam aturcara sesar berganda adalah terhad. Kehadiran sesar berganda dalam aturcara dapat mengurangkan keberkesanan teknik *fault localization* sedia ada untuk mengesan dengan berkesan disebabkan oleh fenomena gangguan sesar. Selain itu, kebanyakan teknik ini tidak dapat menyetempatkan sesar berganda secara serentak dalam senarai kedudukan diagnosis tunggal. Ini telah mendorong penyelidik untuk menggunakan pendekatan seperti penyahpepijatan one-bug-at-a-time (OBA) dan penyahpepijatan selari. Walau bagaimanapun, menggunakan pendekatan penyahpepijatan OBA, ia telah meningkatkan perisian time-to-delivery dan berpotensi membawa kepada lebih banyak sesar semasa pengujian regresi, semasa menggunakan algoritma kelompok k-mean dengan metrik jarak Euclidean untuk mengumpulkan ujian gagal berdasarkan perlaksanaan kesamaan profil dalam pendekatan penyahpepijatan selari, didakwa sebagai bermasalah dan tidak sesuai. Kajian ini adalah bertujuan untuk menjalankan siasatan terhadap pendekatan penyahpepijatan selari yang didakwa bermasalah dan dibandingkan dengan pendekatan penyahpepijatan OBA dan penyahpepijatan selari MSeer dari segi keberkesanan pensetempatan. Selain itu, dua novel teknik *fault localization* berdasarkan teori rangkaian kompleks, iaitu *fault localization* berganda berdasarkan teori rangkaian kompleks (FLCN-M) dan *fault localization* tunggal berdasarkan teori rangkaian kompleks (FLCN-S), dicadangkan untuk memperbaiki keberkesanan pensetempatan, dan untuk membantu pembangun mensetempatkan sesar berganda secara serentak dalam senarai kedudukan

diagnosis tunggal. Teknik-teknik yang dicadangkan menyusun pernyataan cacat-cela berdasarkan keabnormalan kelakuan mereka dan jarak di antara pernyataan aturcara dalam kedua-dua perlaksanaan ujian lulus dan gagal. Di dalam kes di mana pembangun menyemak 70% aturcara dan tidak sepenuhnya mensetempatkan semua sesar berganda dalam senarai kedudukan diagnosis tunggal, dan bukannya terus menggunakan pendekatan penyahpepijatan OBA, pendekatan baru iaitu pengasingan sesar berdasarkan komuniti yang menggunakan algoritma kelompok komuniti rangkaian yang bersifat pecahan, diaplikasikan untuk membantu dalam mengasingkan sesar ke dalam komuniti sesar-fokus (kelompok) berlainan, masing-masing mensasarkan sesar tunggal. Di dalam pendekatan pengasingan sesar yang dicadangkan, pemberat komuniti dan mekanisma pemilihan diperkenalkan untuk membantu dalam mengutamakan komuniti yang paling penting bagi pembangun untuk menyahpijat sesar pada masa yang sama secara selari. Eksperimen yang dijalankan ke atas beberapa aturcara sesar berganda menunjukkan bahawa walaupun pendekatan penyahpepijatan selari yang didakwa bermasalah adalah lebih berkesan berbanding pendekatan penyahpepijatan OBA, ianya tidak berkesan apabila dibandingkan dengan pendekatan penyahpepijatan selari MSeer. Keputusan kajian juga menunjukkan bahawa FLCN-M adalah lebih berkesan dalam mengesan sesar berganda berbanding dengan teknik two similarity coefficient-based (Ochiai coefficient dan Tarantula coefficient) dan teknik *fault localization* yang sedia ada berdasarkan pengukuran pemusatan rangkaian perisian. FLCN-S menunjukkan peningkatan yang ketara dari segi keberkesanan pensetempatan pada aturcara sesar tunggal berbanding dengan teknik-teknik berasaskan Ochiai dan berasaskan Jaccard. Pendekatan pengasingan sesar berdasarkan komuniti menunjukkan peningkatan yang ketara di mana ia melaksanakan dengan lebih baik dari segi keberkesanan pensetempatan berbanding dengan pendekatan penyahpepijatan selari yang didakwa bermasalah dan pendekatan penyahpepijatan selari MSeer. Secara keseluruhannya, teknik dan pendekatan yang

dicadangkan telah menunjukkan peningkatan keberkesanan di dalam kedua-dua aturcara sesar tunggal dan sesar berganda terhadap teknik dan pendekatan garis asas.

Keywords: Teori Rangkaian Kompleks; Perisian Pensetempatan Sesar; Penyahpepijatan Aturcara; Penyahpepijatan Selari; Berganda Sesar.

ACKNOWLEDGEMENTS

First and foremost, I am humbly indebted to Allah Almighty for endowing me with good health and supreme blessing with which the learning I undertook in a mission to seek knowledge. Secondly, I would like to offer my special appreciation to my supervisor Professor Dr. Lee Sai Peck for all her guidance, supervision, compassionate attitude, and consistent encouragement throughout this research work. Prof, Dr. Lee Sai Peck not only provided helpful suggestions and guidance, but has also marvelously fulfilled the responsibility to oversee this research, and guided me towards the successful completion of this thesis. She also gives me the opportunity to expand my professional knowledge, to learn on critical articulation of ideas, and also prepared me for the future challenges.

I am grateful to my research lab colleagues for their continued encouragement and support throughout this research process, I wish them all the best in their future undertakings. Am also grateful to all my friends for providing assistance and cooperation that enabled me to complete this long and fruitful journey.

I would also like to express my gratitude to my family for their endless love and support during my entire life. Firstly, my gratitude goes to my father (Zakari Sadiq Buda) whose tireless sacrifices and support are unforgettable. My gratitude also goes to my beloved mother (Harira Bello) in extension to all my family members for their prayers and support. Without the moral patronage of my family, this thesis would have never been accomplished. No words can express my feelings, so I dedicate this achievement as a gift to my family.

TABLE OF CONTENTS

Abstract iv

Abstrak vi

Acknowledgements ix

Table of Contents x

List of Figures xv

List of Tables xviii

List of Symbols and Abbreviations xxi

List of Appendices xxiii

CHAPTER 1: INTRODUCTION 1

1.1 Background 1

1.2 Motivation 7

1.3 Problem Statement 10

1.4 Research Objectives 12

1.5 Research Questions 13

1.6 Thesis Organization 14

1.7 Chapter Summary 16

CHAPTER 2: LITERATURE REVIEW 17

2.1. Preliminaries 17

2.2. Software Fault Localization Techniques 18

2.2.1 Spectrum-based Fault Localization Technique (SBFL) 18

2.2.2 Statistical-based Fault Localization Technique 23

2.2.3 Model-based Diagnosis Technique 24

2.2.4 Program Slice-based Technique 26

2.2.5 Machine Learning-based Fault Localization Technique 29
2.3. Multiple Fault Localization 30
2.3.1. Fault Localization Interference 31
2.3.2. One-Bug-at-a-Time Debugging Approach (OBA) 32
2.3.3. Parallel Debugging Approach 37
2.4. Complex Network Theory 40
2.5. Research Gap 44
2.6. Chapter Summary 46

CHAPTER 3: RESEARCH METHODOLOGY 47

3.1. Research Process 47
3.2. Investigative Study of the Claimed Problematic Parallel Debugging Approach 50
3.2.1. Parallel Debugging 50
3.2.2. K-means Clustering 53
3.2.3. Similarity Coefficients Metrics 59
3.3. Multiple Fault Localization based on Complex Network Theory (FLCN-M) 61
3.3.1. Network Modeling 66
3.3.2. General Framework 68
3.3.3. A Running Example 69
3.4. Single Fault Localization based on Complex Network Theory (FLCN-S) 71
3.5. Community-based Fault Isolation Approach for Simultaneous Fault Localization 74
3.5.1. Community Clustering Algorithm 74
3.5.2. Shortest-path Betweenness 77
3.5.3. Community Weighting and Selection 81
3.5.4. The Community-based Fault Isolation Approach 83

3.5.5. A Running Example 85
3.6. Chapter Summary 88

CHAPTER 4: EXPERIMENTATION 89

4.1. Data Collection 89
4.2. Experiments 90
4.2.1. Experiment 1: Investigative study of the Claimed Problematic Parallel Debugging Approach 90
4.2.1.1. Subject Programs 91
4.2.1.2. Evaluation Metrics 94
4.2.1.3. Approaches for Cross-comparison 98
4.2.2. Experiment 2: Multiple Fault Localization based on Complex Network Theory (FLCN-M) 98
4.2.2.1. Subject Programs 99
4.2.2.2. Evaluation Metrics 101
4.2.2.3. Techniques for Cross-comparison 103
4.2.3. Experiment 3: Single Fault Localization based on Complex Network Theory (FLCN-S) 104
4.2.3.1. Subject Programs 104
4.2.3.2. Evaluation Metrics 105
4.2.3.3. Techniques for Cross-comparison 106
4.2.4. Experiment 4: Community-based Fault Isolation Approach 107
4.2.4.1. Subject Programs 107
4.2.4.2. Evaluation Metrics 107
4.2.4.3. Approaches for Cross-comparison 108
4.3. Chapter Summary 108

CHAPTER 5: RESULTS AND DISCUSSION 110

5.1. Experiment 1 110

5.1.1. The Effectiveness of the Claimed Problematic Parallel Debugging Approach 111

5.1.2. Cross-Comparison with OBA Debugging Approach 116

5.1.3. Cross-Comparison with MSeer Debugging Approach 123

5.1.4. Result Summary 127

5.2. Experiment 2 129

5.2.1. Effectiveness of FLCN-M on Single-Fault Programs 130

5.2.2. Effectiveness of FLCN-M on SIEMENS-M 131

5.2.3. Effectiveness of FLCN-M on UNIX Real-life Utility Programs 134

5.2.4. Impact of Centrality Measures on the Proposed FLCN-M Technique 135

5.3. Experiment 3 138

5.3.1. Effectiveness of FLCN-S on Siemens Test Suite Programs 138

5.3.2. Effectiveness of FLCN-S on UNIX Real-life Utility Programs 144

5.3.3. Impact of Centrality Measures on the Proposed FLCN-S Technique 146

5.3.4. Overall Observations 149

5.4. Experiment 4 149

5.4.1. Cross-Comparison with P-Ochiai 150

5.4.2. Cross-Comparison with MSeer and P-Ochiai 155

5.4.3. Distance Metrics 159

5.4.4. Result Summary 160

5.5. Chapter Summary 161

CHAPTER 6: CONCLUSION 163

6.1. Summary of Findings in Relation to the Research Objectives 163

6.2. Research Contributions 168
6.3. Limitations of the Study 169
6.4. Future Research Directions 170
6.5. Final Words 171
REFERENCES 172
LIST OF PUBLICATIONS AND PAPERS PRESENTED 185
APPENDIX 187

LIST OF FIGURES

Figure 1.1	:	Effect of multiple faults on fault localization techniques…….....	7
Figure 2.1	:	Coverage data and execution result…….	19
Figure 2.2	:	Directed graph and undirected graph…….....	42
Figure 3.1	:	Process flow of the research activities…….....	48
Figure 3.2	:	Sequential processing of task…….....	51
Figure 3.3	:	Parallel processing of task…….....	51
Figure 3.4	:	Program with all test cases……..	52
Figure 3.5	:	An example of a network consisting of 4 nodes and 4 edges….....	62
Figure 3.6	:	Closeness centrality example for program mid ( )……...	64
Figure 3.7	:	FLCN-M fault localization process……	68
Figure 3.8	:	Network for the multiple-fault program mid ( )……..	70
Figure 3.9	:	Network for the single-fault program mid ( )……..	73
Figure 3.10	:	Calculation of shortest-path betweenness……...	78
Figure 3.11	:	Network with groups of communities…….....	83
Figure 3.12	:	General framework of the proposed approach…….....	85
Figure 3.13	:	Calculation of shortest-path betweenness……...	87
Figure 5.1	:	TDE score-based comparison of the claimed problematic parallel debugging approach with respect to the three coefficients (best case)……	114
Figure 5.2	:	TDE score-based comparison between P-Ochiai and OBA debugging approaches on tcas, replace, and gzip (2-fault versions)…….	120
Figure 5.3	:	TDE score-based comparison between P-Ochiai and OBA debugging approaches on sed and flex (2-fault versions)……	121

Figure 5.4 : TDE score-based comparison between P-Ochiai and MSeer on gzip and grep (3-fault versions)……..... 126

Figure 5.5 : Effectiveness comparison between FLCN-M and other fault localization techniques on Single-Fault Programs (Siemens test suite)…….. 130

Figure 5.6 : IDE score-based comparison on 2-fault versions…….. 132

Figure 5.7 : IDE score-based comparison on 3-fault versions…….. 132

Figure 5.8 : IDE score-based comparison on 4-fault versions…….. 133

Figure 5.9 : IDE score-based comparison on 5-fault versions…….. 133

Figure 5.10 : IDE score-based comparison on between FLCN-M, Tarantula, Ochiai, and SNCM on gzip program.……. 134

Figure 5.11 : IDE score-based comparison between FLCN-M, Tarantula, Ochiai, and SNCM on sed program.……... 135

Figure 5.12 : SIEMENS-M (Degree centrality of program statement and its correlation with failure)…….. 136

Figure 5.13 : Gzip (Degree centrality of program statement and its correlation with failure)……... 137

Figure 5.14 : Sed (Degree centrality of program statement and its correlation with failure)……..... 137

Figure 5.15 : EXAM score-based Comparison between FLCN-S and the Baseline techniques on tcas, print_tokens, and print_tokens2…… 141

Figure 5.16 : Overall effectiveness comparison on Siemens test suite…… 142

Figure 5.17 : EXAM score-based comparison between FLCN-S and Ochiai on gzip, and sed.…….. 145

Figure 5.18 : Degree centrality correlation with failure for UNIX real-life utility programs…….. 147

Figure 5.19 : Degree centrality correlation with failure for Siemens test suite programs……. 148

Figure 5.20 : TDE score-based comparison between the proposed approach and P-Ochiai on gzip, sed, and flex (2-fault versions)……... 153

Figure 5.21 : TDE score-based comparison between the proposed approach with P-Ochiai and MSeer on gzip and grep (3-fault versions)……. 158

LIST OF TABLES

Table 2.1	:	Notations widely used in suspiciousness calculation..…….....	21
Table 3.1	:	An example of failed tests execution with 7 statements and 8 failed test cases…….....	57
Table 3.2	:	First k-means clustering iteration (Iteration 1)……..	58
Table 3.3	:	Similarity coefficient metrics…..…….....	60
Table 3.4	:	A multiple-fault program with tests execution……..	67
Table 3.5	:	Network construction sequence…….....	67
Table 3.6	:	Localization result of program mid ( ) with multiple faults……	71
Table 3.7	:	Program mid ( ) with a single fault…….	72
Table 3.8	:	Localization result of program mid ( ) with single fault…….	73
Table 3.9	:	A program with tests execution…….....	86
Table 3.10	:	Final edge-betweenness score for each edge in the network……..	87
Table 4.1	:	Experimental subject programs……	91
Table 4.2	:	Summary of Siemens test suite programs……..	100
Table 4.3	:	UNIX real-life utility programs…….....	101
Table 5.1	:	Average number of statements examined (best case)…….....	112
Table 5.2	:	The confidence with which it can be claimed that P-Ochiai is more effective than P-Naish and P-Jaccard (best cases)…….....	115
Table 5.3	:	Average number of statements examined (best case)…….	117
Table 5.4	:	Average number of statements examined (worst case)……..	117
Table 5.5	:	The confidence with which it can be claimed that P-Ochiai is more effective than OBA approach (best cases)…….	122
Table 5.6	:	The confidence with which it can be claimed that P-Ochiai is more effective than OBA approach (worst cases)……...	122

Table 5.7 : Average number of statements examined (best case)……. 124

Table 5.8 : Average number of statements examined (worst case)…….. 125

Table 5.9 : The confidence with which it can be claimed that MSeer is more effective than P-Ochiai (best & worst cases)…….. 127

Table 5.10 : Cumulative number of statements examined to locate faults for each program in Siemens test suite (best & worst cases)…… 139

Table 5.11 : The confidence with which it can be claimed that FLCN-S is more effective than Ochiai and Jaccard on Siemens test suite programs (best & worst cases)……..... 142

Table 5.12 : Cumulative number of statements examined by FLCN-S and Ochiai (best & worst cases)……... 144

Table 5.13 : The confidence with which it can be claimed that FLCN-S is more effective than Ochiai (best & worst cases)……. 146

Table 5.14 : Average number of statements examined (best case)……. 151

Table 5.15 : Average number of statements examined (worst case)…….. 151

Table 5.16 : The confidence with which it can be claimed that the proposed approach is more effective than P-Ochiai (best cases)……. 154

Table 5.17 : The confidence with which it can be claimed that the proposed approach is more effective than P-Ochiai (worst cases)……. 154

Table 5.18 : Average number of statements examined using the proposed approach, MSeer, and P-Ochiai (3-fault versions)……. 156

Table 5.19 : The confidence with which it can be claimed that the proposed approach is more effective than MSeer and P-Ochiai (3-fault versions)…… 158

Table 5.20 : Average number of statements examined using the proposed approach, MSeer, and P-Ochiai (5-fault versions)…… 160

LIST OF ABBREVIATIONS

BMPS	:	Bounded Debugging via Multiple Predicate Switching
CLPS-MFL	:	Concept Lattice of Program Spectrum for Effective Multiple Fault Localization
CC	:	Closeness Centrality
DNN	:	Deep Neural Network
DMS	:	Diversity Maximization Speedup
DC	:	Degree Centrality
DD	:	Delta Debugging
FLCN-S	:	Single Fault Localization Based on Complex Network Theory
FLCN-M	:	Multiple Fault Localization Based on Complex Network Theory
IVMP	:	Interesting Value Mapping Pair
IDE	:	Incremental Developer Expense
MBD	:	Model-based Diagnosis
NIST	:	National Institute of Standards and Technology
NN	:	Nearest Neighbor
OBA	:	One-bug-at-a-time
PPDG	:	Probabilistic Program Dependence Graph
PBC	:	Parameter-based Combination
PDG	:	Program Dependency Graph
SIR	:	Software Infrastructure Repository
SBFL	:	Spectrum-based Fault Localization
SAT	:	Satisfiability-based Formula Verification Technique
SFL	:	Software Fault Localization
SNCM	:	Software Network Centrality Measure

TDE	:	Total Developer Expense
WWW	:	World Wide Web

LIST OF APPENDICES

Appendix A : Results for FLCN-S on the Single Fault Experiment (Experiment 3)……..... 187

Appendix B : Results for FLCN-M on the Multiple Fault Experiment (Experiment 2)……..... 199

CHAPTER 1: INTRODUCTION

1.1 Background

Software has become part of our daily lives. It is integrated into practically everything we do and devices we are dependent on (Wong et al, 2016). The dependency and influence of software come with increasing challenges in maintaining software quality and complexity reduction. Complexity is one of the major contributions of software faults, which increase software failures (Vessey, 1985). A 2002 report by the National Institute of Standards and Technology (NIST) shows the impact of software errors on the U.S. economy which was estimated to cost about $59.5 billion annually (NIST, 2002). Therefore, effective ways of finding and fixing faults are of eminent advantage from an economic perspective. Debugging is the process of locating and correcting program faults. Generally, debugging activities are twofold, fault localization and fault repair. The former is regarded as one of the most tedious and costly activities in the debugging process (J. A. Jones, Harrold, & Stasko, 2002). Fault localization is undoubtedly vital in maintaining software quality, as the faster a fault location is identified, the faster it can be neutralized (Abreu, Zoeteweij, & Van Gemund, 2007). Early fault identification improves software availability and reduces software cost.

In the last decades, various fault localization techniques had been proposed with competing ways of effective fault identification (DiGiuseppe & Jones, 2015; J. A. Jones & Harrold, 2005; J. A. Jones et al., 2002; Lamraoui & Nakajima, 2016; B. Liu, Nejati, Briand, & Bruckmann, 2016; Sun, Peng, Li, Li, & Wen, 2016; W. E. Wong, Debroy, Gao, & Li, 2014; W. Zheng, Hu, & Wang, 2016). In general, a fault localization technique assigns suspicious scores to program statements that signify a statement's degree of association with failure. A ranking list of all program statements in descending order of their suspicious scores is generated to aid software developers in checking fewer program

statements and thus facilitating the localization process. A good fault localization technique should rank faulty program statements at the top or closer to the top in the ranking list. The faster a faulty statement can be found, the more effective is the fault localization technique (W. E. Wong et al., 2014). One of the most promising fault localization techniques is spectrum-based fault localization technique (SBFL). This technique identifies suspicious program statements using program execution information (i.e. program spectra and test results). With this information, behavioral abnormalities are analyzed to help identify program locations that are more prone to error (Abreu et al., 2007). Most of the existing techniques are proven to be helpful in facilitating software development and maintenance process especially on single-fault programs (Abreu & Zoeteweij, 2006). However, although empirical studies revealed that failure in programs can be caused by multiple faults (DiGiuseppe & Jones, 2011b; James A Jones, Bowring, & Harrold, 2007), most existing techniques localize faults based on the assumption that a program has a single fault (W. Zheng et al., 2016).

Consequently, this presumption adversely impacts the effectiveness of fault localization due to the possibility of having more than one fault in a faulty program (DiGiuseppe & Jones, 2015). Principally, this is due to fault interference, a phenomenon which plays a major role in the reduction of the effectiveness of fault localization techniques in the context of multiple faults. Fault interference phenomenon occurs when a test case that failed in the presence of single fault, passes when many faults are active; while a test case that passed in the presence of multiple faults, fails when a single fault is active. Multiple fault localization refers to the process of localizing multiple faults in a software program. A significant number of techniques have been proposed to localize multiple faults efficiently and effectively (James A Jones et al., 2007; Lamraoui & Nakajima, 2016; Sun et al., 2016; W. E. Wong et al., 2014). Efforts have been made to further isolate independent faults into different clusters for simultaneous localization

(DiGiuseppe & Jones, 2012a; James A Jones et al., 2007; Liblit, Naik, Zheng, Aiken, & Jordan, 2005). However their effectiveness is still not optimal, and the techniques are not able to localize multiple faults simultaneously in a single diagnosis rank list. Hence, an effective fault localization technique is needed to localize multiple faults effectively in a single diagnosis rank list or in a single debugging iteration (i.e. neutralizing all faults in a single debugging iteration).

Furthermore, most of the existing fault localization techniques used on multiple-fault programs localize faults using one-bug-at-a-time debugging approach (OBA) (DiGiuseppe & Jones, 2011b; J. A. Jones & Harrold, 2005; W. E. Wong et al., 2014). OBA debugging approach is the process whereby a developer is expected to find a fault, fix it, and then re-test the program to find the remaining faults. This process is performed iteratively until all the faults are found and fixed. This results in the approach creating more faults during regression testing and also increasing software time-to-delivery (Xue & Namin, 2013). Many empirical studies have shown the downside of using the OBA debugging approach in programs with multiple faults (DiGiuseppe & Jones, 2011a, 2011b, 2015) because the labor cost to localize and fix faults, and the time required to produce a failure-free program can be very high. Due to this issue, parallel debugging approach has been utilized by various studies (DiGiuseppe & Jones, 2012b; Huang, Wu, Feng, Chen, & Zhao, 2013; James A Jones et al., 2007). Parallel debugging approach aids in the isolation of distinct faults into separate *fault-focused* clusters for multiple developers to debug the faults simultaneously in parallel, ideally to reduce debugging cost and time. Parallel debugging workflow was first proposed by Jones et al. (James A Jones et al., 2007), with the idea of reducing debugging cost in the context of multiple faults. In parallel debugging, failed tests execution are partitioned into clusters that target a single fault each. These clusters are coined *fault-focused* clusters. Therefore, to create a specialized test suite that might target a single fault, each *fault-focused* cluster will be

combined with all available passed test cases. Finally, the specialized test suites will be assigned to multiple developers to debug the faults in parallel. The result of their study showed that parallel debugging approach can help in reducing debugging cost and decrease the time to deliver a failure-free program.

Hence, the most important component of parallel debugging approach is clustering, particularly on how to obtain a good clustering on failed test cases that target single faults especially when knowing in real-life cases, developers do not know the exact number of faults in a faulty program. Many clustering algorithms were used in recent years such as *k*-means clustering algorithm, hierarchical clustering algorithm, and *k*-medoids clustering algorithm (R. Gao & Wong, 2017; Högerle, Steimann, & Frenkel, 2014; James A Jones et al., 2007; Steimann & Frenkel, 2012; Yabin Wang, Gao, Chen, Wong, & Luo, 2014).

In most of these studies, failed tests are grouped based on their execution profile similarity. In a study by Huang et al., the researchers utilized *k*-means and hierarchical clustering algorithms to cluster failed tests execution based on the similarity of their execution profile and found that *k*-means is effective for isolating faults in fault localization (Huang et al., 2013). Yet, studies such as (R. Gao & Wong, 2017; C. Liu, Zhang, & Han, 2008) suggest that such grouping is problematic because a fault can be triggered in different ways. Secondly, the number of clusters is estimated based on the number of failed test cases, this was however also questioned by Gao et al. because there is no clear correlation between the number of failed test cases and the number of faults is a faulty program (R. Gao & Wong, 2017). Thirdly, the distance metrics that measure the *due-to* relationship between failed tests execution which gravely determines which cluster a given failed tests will fall under, is also very important and critical. Distance metrics such as Euclidean distance, Jaccard distance, Hamming distance have been used in fault localization research domain (Huang et al., 2013; James A Jones et al., 2007). These

metrics were claimed to be inappropriate to use for software fault localization in measuring the *due-to* relationship between failed test cases (R. Gao & Wong, 2017). Hence, the combination of these two component (clustering algorithm and distance metric) have an impact on both the clustering results and the fault localization effectiveness. Despite these issues, there are no studies that investigate the claimed problematic parallel debugging approach that uses the clustering algorithms that groups failed test cases based on their execution profile similarity, estimate the number of clusters based on the number of failed test cases, and utilize distance metric such as Euclidean distance metric in localizing multiple faults.

This work aims to conduct an investigative study of the claimed problematic parallel debugging approach using the existing parallel workflow as utilized by previous studies (James A Jones et al., 2007; W. E. Wong, Debroy, Golden, Xu, & Thuraisingham, 2012). Firstly, an investigative study on the effectiveness of the claimed problematic parallel debugging approach that makes use of a *k*-means clustering algorithm (that groups tests execution based on their execution profile similarity) with Euclidean distance on three well-known similarity coefficient-based fault localization techniques is conducted. Secondly, a cross-comparison between the claimed problematic parallel debugging approach and OBA debugging approach is conducted in terms of localization effectiveness. Additionally, a comparative study is conducted between the claimed problematic parallel debugging approach and MSeer parallel debugging approach proposed by Gao et al. (R. Gao & Wong, 2017).

Furthermore, two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), are proposed to improve localization effectiveness in programs with single and multiple faults and to

aid developer to localize multiple faults simultaneously in a single diagnosis rank list (single debugging iteration). The proposed techniques rank statements based on their behavioral abnormalities and distance between statements in both passed and failed tests execution. Two graph-based centrality measures, namely degree centrality and closeness centrality, are used for fault diagnosis, and a new ranking formula is proposed to calculate the suspiciousness of program statements. Furthermore, in a situation where a developer has checked 70% of the program statements and cannot localize all the multiple faults in a single debugging iteration, instead of the developer utilizing the OBA debugging approach, a newly proposed community-based fault isolation approach that makes use of a divisive network community clustering algorithm will be applied to aid in the effective isolation and localization of multiple faults. A community weighting and selection mechanism is introduced in the proposed fault isolation approach to aid in prioritizing highly important communities to developers to debug the faults simultaneously in parallel. Based on the experiments performed on various single-fault and multiple-fault programs, the proposed techniques' and approach shows significant improvement in comparison with the baseline techniques and approaches in terms of localization effectiveness.

In this thesis, complex network theory is used for fault localization for the following reasons. Firstly, complex network theory has largely shown to be an applicable theory in the field of science and it has been effectively used to solve several problems in research areas such as physics (Albert & Barabási, 2002), biology (Dorogovtsev & Mendes, 2003), social network (L. Freeman, 2004), and software engineering (Chong & Lee, 2015). Secondly, complex network has the ability to help researchers understand complex systems. For instance, it can be used to identify important statements and their correlation with faults in software programs (Zhu, Yin, & Cai, 2011). Lastly, in the context of fault

localization, complex network theory has the ability to aid in the identification and localization of faulty program statements and statements that are related to failure.

1.2 Motivation

This research was triggered by the negative impact of fault localization techniques effectiveness in the context of multiple faults. Various studies have shown that the existence of multiple faults in a software program reduces the efficacy of the existing fault localization techniques to locate faults effectively (DiGiuseppe & Jones, 2011b; Xue & Namin, 2013). A program with two faults is illustrated as an example to show the effect of multiple faults on the effectiveness of SBFL techniques. The example shown in Figure 1.1 has two faults "fault 1" and "fault 2" and 4 test cases (t_1, t_2, t_3, t_4) in which t_1 and t_2 are passed tests while t_3 and t_4 are failed tests. If a statement execution is labeled as 1, that means the statement is executed by the test case in that test run. If a statement is labeled as 0, that means the statement is not executed by the test case in that test run. For the test result of each test case, 0 means the test case has passed while 1 means the test case has failed.

	t1	t2	t3	t4	Suspiciousness (t_1,t_2,t_3,t_4)	Suspiciousness (t_1,t_2,t_4)	Suspiciousness (t_1,t_2,t_3)
1: if (b) {	1	1	1	1	0.70	0.60	0.60
2: *fault 1;*	1	0	1	0	0.50	0	0.70
3: } else {	0	1	0	1	0.50	0.70	0
4: *fault 2;*	0	1	0	1	0.50	0.70	0
5: }	1	1	1	1	0.70	0.60	0.60
Test Result	0	0	1	1			

Figure 1.1: Effect of multiple faults on fault localization techniques

The two test cases t_3 and t_4 both failed due to a different fault because both test cases execute different faults. Test case t_3 only executes "fault 1", while test case t_4 only executes "fault 2". This situation will make fault localization difficult, especially when a developer wants to localize all the faults simultaneously in a single debugging iteration. The last 3 columns in Figure 1.1 list down the suspicious scores of the program statements based on the selected test suite used for fault localization. Program statements with higher suspicious score have a higher likelihood of being faulty. In this example, Ochiai similarity coefficient-based fault localization technique proposed by Abreu et al. (Abreu et al., 2007) is used for the computation of the suspicious score. Ochiai coefficient, S_S, is calculated as depicted in Equation 1.1.

$$S_s = \frac{Ncf}{\sqrt{(Ncf + Nnf) \times (Ncf + Ncs)}} \qquad (1.1)$$

Ncf denotes the number of failed test cases that cover a statement, and *Nnf* denotes the number of failed test cases that do not cover a statement, while *Ncs* denotes the number of passed test cases that cover a statement. When all test cases are considered for suspicious score computation as shown in the sixth column, the suspicious score of statement 2, statement 3, and statement 4 are the same (*0.50%*) while the non-faulty statements (statement 1 and statement 5) have a very high suspicious score.

As a result, this makes the Ochiai metric ineffective in localizing the two faults "fault 1" and "fault 2" if all tests are considered. Nonetheless, if t_3 is removed from the test suite as shown in the seventh column, a developer can effectively localize "fault 2" which was given a high suspicious score (*0.70%*). On the other hand, if t_4 is removed from the test suite (eighth column), "fault 1" can be effectively localized by the technique because "fault 1" has the highest suspicious score with (*0.70%*). Therefore, this scenario illustrates that with two faults in a faulty program, utilizing all the available test cases might reduce

the effectiveness of a fault localization algorithm and a developer cannot localize all the faults in a single debugging iteration. Hence, the example shows that by isolating test cases, a debugger can target and localize each fault separately.

However, isolating faults into many separate groups can increase debugging cost as well, and the effectiveness solely depends on the accuracy of clustering (Jeffrey, Gupta, & Gupta, 2009). Fault localization gets even more difficult if more faults are present in a program (J. A. Jones & Harrold, 2005). This example illustrates in detail the ineffectiveness of SBFL techniques in a program with multiple faults when utilizing all test cases. Hence, localizing all the faults in a single diagnosis rank list is not possible in this scenario. Similarly, utilizing the OBA debugging approach and parallel debugging approach does not solve the problem completely, whereby using the former, there is a possibility of creating more faults during regression testing and also increasing the software time-to-delivery due to the fact that many debugging iterations are needed to neutralize all faults. Using the latter, the most important component is clustering (i.e. how to perform good clustering on failed test cases that target single faults) and distance metric (that helps in measuring the *due-to* relationship between failed test cases) which gravely determines which cluster a given failed test will fall under. Most of the existing clustering algorithms and distance metrics used in existing fault localization studies are deemed to be problematic and inappropriate (R. Gao & Wong, 2017; C. Liu et al., 2008).

Hence, a fault localization technique that can effectively localize multiple faults simultaneously in a single diagnosis rank list is of great importance. Additionally, where many faults exist in a program that cannot be all localized in a single debugging iteration, a new approach that can efficiently isolate faulty program statements into distinct *fault-focused* clusters will further aid in a more effective localization.

1.3 Problem Statement

Advancement in software development with the increasing complexity of software programs has led to the increase in software failure in recent years (W. E. Wong et al., 2016). This also results in more faults during software development that adversely causes failure of software programs, which takes a toll on software quality due to the lack of software conformance to its requirements (Zakari, Lawan, & Bekaroo, 2016). Moreover, 50% - 80% of development and maintenance cost is spent in the debugging process which is also considered as one of the tedious, time-consuming, and costly activities in software testing (Collofello & Woodfield, 1989). This activity involves failure detection, fault localization, and fault repair. Fault localization has received much research attention in the past decades, notably because the process tends to be difficult when it is used manually (Agrawal, DeMillo, & Spafford; Hennessy, 1982; Rosenblum, 1995). The manual fault localization techniques make the process slow and costly, especially when debugging large-scale software programs that have thousands or millions of lines of code (Yu, Jones, & Harrold, 2008). This has driven the interest of many researchers to automate the process which paves the introduction of various automated fault localization techniques (Baudry, Fleurey, & Le Traon, 2006; Richard A. DeMillo, Pan, & Spafford, 1997; J. A. Jones & Harrold, 2005; Renieres & Reiss, 2003; A. X. Zheng, Jordan, Liblit, & Aiken, 2003).

However, earlier studies localize faults based on the assumption that a program has a single fault, which in reality is not the case (J. A. Jones & Harrold, 2005; W. Zheng et al., 2016). Empirical studies revealed that when a program fails, the failure is not only caused by a single fault but can rather be caused by multiple faults (James A Jones et al., 2007; A. X. Zheng, Jordan, Liblit, Naik, & Aiken, 2006). This presumption of the existing techniques during fault localization has affected their effectiveness on a great margin.

Moreover, due to the complex relationship between fault and failure specifically in the existence of multiple faults, existing techniques find it hard to localize multiple faults simultaneously, whereby test cases that failed in the existence of single fault could pass in the existence of multiple faults, and a test case that passed in the presence of multiple faults could fail when a single fault is active.

Furthermore, most of the existing fault localization techniques used on multiple-fault programs localize faults using the OBA debugging approach (DiGiuseppe & Jones, 2011b; J. A. Jones & Harrold, 2005; W. E. Wong et al., 2014), which creates more faults during regression testing (DiGiuseppe & Jones, 2015). This has led to software developers perceiving that those faults are not being localized and fixed. Due to this reason, the OBA debugging approach increases the time-to-delivery of the software program and directly reduces the effectiveness of the existing fault localization techniques (DiGiuseppe & Jones, 2015). As a result, fault localization techniques for multiple-fault programs are needed to localize faults simultaneously in a single diagnosis rank list (i.e. neutralizing all faults in a single debugging iteration).

Also, parallel debugging approach that performs clustering on failed test cases based on their execution profile similarity is claimed to be problematic (R. Gao & Wong, 2017). In a study by Huang et al., the researchers utilized *k*-means and Hierarchical clustering algorithm to cluster failed tests execution based on the similarity of their execution profile and found that *k*-means is effective for isolating faults in fault localization (Huang et al., 2013). Such a method of clustering is also used by other multiple fault localization studies (Högerle et al., 2014; Huang et al., 2013; Steimann & Frenkel, 2012). Yet, studies such as (R. Gao & Wong, 2017; C. Liu et al., 2008) suggest that such grouping is problematic because a fault can be triggered in different ways. Excluding some failed test cases only because their tests execution coverage differs from other tests will reduce the

effectiveness of fault localization. These studies also argued that such tests representation will result in poor fault localization results due to the poor clustering method utilized. Moreover, distance metrics such as Euclidean distance, Jaccard distance, and Hamming distance were claimed to be inappropriate to use for software fault localization in measuring the *due-to* relationship between failed test cases to determine which cluster a failed test will fall under (R. Gao & Wong, 2017).

Despite these issues, there are no studies that investigate the claimed problematic parallel debugging approach that uses clustering algorithms such as *k*-means (that groups failed test cases based on their execution profile similarity) with a distance metric such as Euclidian distance in localizing multiple faults. In addition, the prevailing problem of clustering accuracy for fault isolation in parallel debugging approach when localizing multiple faults simultaneously, has called for the need of a new clustering method for fault isolation.

1.4 Research Objectives

The aim of this research is to propose two new fault localization techniques based on complex network theory that will aid in the simultaneous localization of single and multiple faults in a single diagnosis rank list as well as a new community-based fault isolation approach that will aid in isolating and localizing multiple faults simultaneously in parallel. The main research objectives for this study are:

i. To investigate the existing parallel debugging approach used in localizing multiple faults in terms of localization effectiveness in comparison with other debugging approaches.

ii. To propose two novel fault localization techniques for single-fault and multiple-fault programs based on complex network theory.

iii. To propose a new community-based fault isolation approach to aid in the effective isolation and localization of multiple faults simultaneously in parallel.

iv. To evaluate the proposed fault localization techniques and the proposed approach by comparing them with the baseline techniques and approaches in terms of localization effectiveness.

1.5 Research Questions

To achieve the research objectives of this research, research questions are formulated to help the researcher in adhering to the research objectives. The following are the list of Research Questions (RQs) for this study:

i. RQ1: What are the existing studies utilizing parallel debugging approach used in localizing multiple faults?

   RQ1.1: Do the existing studies utilizing parallel debugging approach group failed test cases based on their execution profile similarity?

   RQ1.2: Do the existing studies utilizing parallel debugging approach provide a good method for estimating the number of clusters?

   RQ1.3: What are the distance metrics used in the existing studies that utilize parallel debugging approach?

ii. RQ2: How can complex network theory be used to improve localization effectiveness on both single-fault and multiple-fault programs?

iii. RQ3: How to localize multiple faults simultaneously in a single diagnosis rank list?

iv. RQ4: How to effectively isolate and localize multiple faults simultaneously in parallel based on complex network theory?

v. RQ5: What is the performance of the proposed fault localization techniques and the proposed approach in comparison with the baseline techniques and approaches in terms of localization effectiveness?

1.6 Thesis Organization

This section gives a brief overview of the whole thesis. The thesis is organized into six chapters in aiming to provide a comprehensive study of this research.

Chapter 1 - Introduction

This chapter starts by providing the background and motivation of this research followed by the problems this research intends to address. The chapter also outlines the research aim, research objectives, and research questions of the study.

Chapter 2 – Literature Review

This chapter highlights some of the basic terminologies that are vital in the study of faults to failure relationship between program entities in software programs. Some of the most prominent software fault localization techniques in the research domain are also presented. The chapter further discusses multiple fault localization, fault interference, and the two main approaches used in debugging multiple faults. Studies utilizing these approaches to localize multiple faults are highlighted in detail. The chapter also gives a basic background literature on complex network theory and some related work on its

application in various research domains including software engineering. The justification of using complex network theory for fault localization is also given. Lastly, the gaps in research and the need for this research is presented.

Chapter 3 – Research Methodology

This chapter discusses the general methodology applied in carrying this research activity. Each of the stages of research is discussed elaborately starting with literature review followed by problem analysis, methodology, and finally evaluation and validation of the proposed techniques and approach. Furthermore, the methodology of the investigative study of the claimed problematic parallel debugging approach is highlighted. The two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), are presented in detail. Lastly, the new community-based fault isolation approach is also presented.

Chapter 4 – Experimental Setup

This chapter discusses the different experimental setups carried out for the implementation and validation of the investigative study of the claimed problematic parallel debugging approach, the two proposed techniques, and the proposed approach. The data collection process is also highlighted. The subject programs, evaluation metrics, and the baseline techniques and approaches for cross-comparison used in each experiment are also detailed.

Chapter 5 – Results and Discussion

This chapter presents the experimental results and discussion of the different experiments carried out in the research. The cross-comparisons between all the baseline techniques and approaches are also provided.

Chapter 6 - Conclusion

This chapter summarizes the research findings and highlights the identified research limitations of the study. The thesis contributions are also highlighted. Finally, recommendations for future work are also provided.

1.7 Chapter Summary

This chapter presented the background and motivation of this research work, and the problems this research intends to address have been defined. The chapter also outlined the research aim, research objectives, and research questions of the study. In conclusion, this chapter presents the scope of this study. The next chapter gives an overview of the literature on software fault localization research domain.

CHAPTER 2: LITERATURE REVIEW

In this chapter, the literature on software fault localization is presented. The chapter starts by highlighting the basic terminologies that are vital in the study of faults to failure relationship between program entities in software programs. Furthermore, some of the most prominent software fault localization techniques in the research domain are presented. The chapter also discusses multiple fault localization, fault interference, and the two main debugging approaches used in debugging multiple faults. Furthermore, the chapter briefly highlights literature on complex network theory with related works. Lastly, the gaps in research and the need for this research is presented.

2.1. Preliminaries

In this section, some basic terminologies as defined in (Avizienis et al., 2004) are provided to facilitate understanding.

i. A failure occurs when a software system gives an unexpected output rather than the anticipated correct output.

ii. An error is a condition caused by a human action in a software system that can lead to failure.

iii. A fault which is also known as a bug is an underlying cause of an error.

iv. Diagnosis is the process of locating faults that are the root cause of the detected errors. Therefore, error detection is a precondition for diagnosis. For a program to be diagnosed, it has to contain a set of statements which are executed using test cases that are either passed or failed. This activity is recorded in terms of program spectra (Harrold et al., 1998).

v. Program spectra data is collected at run-time which consists of the collective tests execution for different components of a software program.

These terminologies are vital in the study of fault to failure relationship between program entities.

2.2. Software Fault Localization Techniques

Software fault localization is an active area of research for the past two decades. Various state-of-the-art techniques have been proposed to localize faults effectively. In this section, some of the most prominent software fault localization techniques in the research area are presented.

2.2.1 Spectrum-based Fault Localization Technique (SBFL)

One of the most promising debugging techniques is spectrum-based fault localization technique (SBFL) which works at the statements level. SBFL computes statements suspicious score using the information gathered from software testing process, such as program spectra and test results of passed and failed test cases (J. A. Jones et al., 2002; Perez, Abreu, & van Deursen, 2017). Program spectra is a pool of data which gives a clear view of the dynamic behavior of a software program (Thomas et al., 1997). In general, program spectra records the run-time profiles of various program entities (i.e. statements, branches, paths, and blocks) for test cases of a given test suite (Cousin, 1986). Furthermore, test results (passed/failed) of test cases are an essential information for fault localization. When combined with a program spectra, the resulting outcome will give developers a hint on the program entities that are more likely to be related to failure or contained faults (Xie et al., 2013). Practically, when an execution failed, the failure paths

are identified to be more likely to contain faults while passed executions are regarded to be less prone to contain faults. Figure 2.1 illustrates the important information required by SBFL. The collected data (as discussed above) are virtually represented. Figure 2.1 illustrates a coverage matrix and a test result vector. For the matrix and test result vector with the entry of (*i, j*), the matrix is 1 if test case *i* covers statement *j*, and 0 otherwise. Moreover, an entry in the test result vector is 1 if the result of test case *i* is failed, and 0 if the result is passed. Each row of the coverage matrix reveals the statements that are covered by the corresponding test cases, and each column shows the coverage vector of the corresponding program statement.

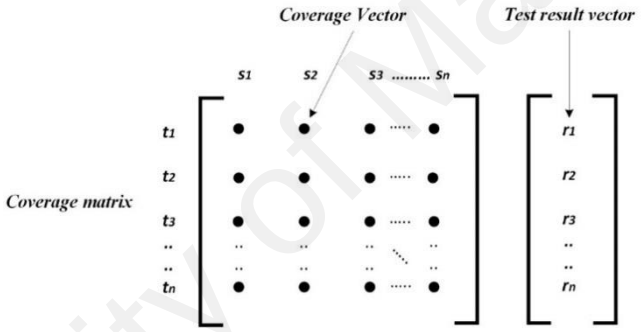


Figure 2.1: Coverage data and execution result

In the early days, researchers used failed tests execution alone in locating faults for SBFL (Agrawal, De Millo, & Spafford, 1991; Korel, 1988; Bogdan Korel & Janusz Laski, 1988). However, this practice was later shown to be ineffective in locating program faults (Agrawal et al., 1995). Studies that use both passed and failed test cases have shown to achieve better results (Abreu & Zoeteweij, 2006; J. A. Jones & Harrold, 2005; Neelofar Neelofar, Naish, Lee, & Ramamohanarao, 2017; E. Wong, Wei, Qi, & Zhao, 2008). Renieres and Reiss proposed a technique named nearest neighbor, which produces a suspiciousness report of program statements by measuring the distance between a failed test and a passed test that is more similar to the failed one (Renieres & Reiss, 2003). In

SBFL, one measures the similarity between the test result vector and the coverage matrix of each statement. This similarity is quantified and measured by a similarity coefficient-based fault localization technique. One of the most popular similarity coefficient-based fault localization techniques is Tarantula (J. A. Jones & Harrold, 2005), which exploits the program spectra of statements execution to identify faults location in a software program. Tarantula assigns a suspicious score to each program statement based on the likelihood of it containing faults, then a developer will be tasked to check the statements in descending order of their suspicious scores to identify the location of faults. Tarantula has shown to be one of the most effective software fault localization techniques. Tarantula coefficient, S_T, is calculated as depicted in Equation 2.1.

$$S_T = \frac{\frac{Ncf}{Ncf + Nuf}}{\frac{Ncf}{Ncf + Nuf} + \frac{Nuf}{Nuf + Nus}} \quad (2.1)$$

where *Ncf* denotes the number of failed test cases that cover a statement, *Nuf* denotes the number of failed test cases that do not cover a statement and *Nus* denotes the number of successful test cases that do not cover a statement. An earlier study shows that Tarantula guides a developer to the location of faults by examining a lesser amount of code in comparison to other fault localization techniques such as set union, nearest neighbor, and cause transition (Cleve & Zeller, 2005; J. A. Jones & Harrold, 2005).

However, other coefficients have been found in recent years that surpass Tarantula in terms of effectiveness at fault localization (E. Wong et al., 2008; W. E. Wong & Qi, 2009). For example, the Ochiai similarity coefficient-based fault localization technique is regarded to be more effective than Tarantula (Abreu & Zoeteweij, 2006). Ochiai coefficient, S_S, is calculated as depicted in Equation 2.2.

$$S_s = \frac{Ncf}{\sqrt{(Ncf + Nnf) \times (Ncf + Ncs)}} \qquad (2.2)$$

Ncf denotes the number of failed test cases that cover a statement, and *Nnf* denotes the number of failed test cases that do not cover a statement, while *Ncs* denotes the number of passed test cases that cover a statement. Notations that are widely used in suspiciousness calculation are highlighted in Table 2.1.

Table 2.1: Notations widely used in suspiciousness calculation

Notation	Description
N	Number of test cases
N_f	Number of failed test cases
N_s	Number of passed test cases
N_{cf}	Number of failed test cases that cover a statement
N_{cs}	Number of passed test cases that cover a statement
N_c	Number of test cases that cover a statement
N_{uf}	Number of failed test cases that cannot cover a statement
N_{nf}	Number of failed test cases that do not cover a statement
N_{us}	Number of passed test cases that cannot cover a statement
N_u	Number of test cases that cannot cover a statement

Naish et al. proposed two SBFL techniques which are O and O^P (Naish, Lee, & Ramamohanarao, 2011). The former is built for programs with a single fault, while the latter is for programs with multiple faults. The result of the study showed that O and O^P are more effective in localizing faults than Tarantula. In order to improve the diagnosis accuracy of SBFL techniques, Shu et al. proposed a fault localization method based on statement frequency (Shu et al., 2016). The statement frequency information of each statement in the software program is used to localize faults. The study showed that the proposed approach outperforms Tarantula in terms of stability and effectiveness respectively. Recently, Wong et al. proposed a new fault localization method named DStar (Wong et al., 2014). The technique is a modified form of *Kulczynski* similarity

coefficient (Choi, Cha, & Tappert, 2010), which has been shown to be very effective and surpasses various fault localization techniques in terms of effectiveness in locating faults.

Researchers in (Abreu et al., 2007; Le, Thung, & Lo, 2013) conducted a comparison study of SBFL techniques. They concluded that SBFL techniques' performance varies based on the debugging scenarios they were applied to, whereby a coefficient can be more effective in a given scenario and less effective in other scenarios (Yoo et al., 2014). Therefore, there is no coefficient that can outperform all others under every scenario.

De Souza et al. proposed a technique to contextualize code inspection to provide guidance during fault localization and improve localization effectiveness of SBFL techniques in localizing fault in the first generated suspiciousness list (De Souza et al., 2018). The result shows that the technique is useful in guiding developers to fault locations and improves localization effectiveness. Another study by Kim et al. proposed a variable centric technique to enhance the performance of existing SBFL techniques (Kim, Kim, & Lee, 2018). The technique extracts suspicious variables and uses them to generate a suspicious ranked list. The result shows that their proposed technique outperforms existing similarity coefficient techniques.

Landsberg et al. improve the effectiveness of SBFL technique by introducing a new method that generates a viable and efficient test suite for effective fault localization (Landsberg, Sun, & Kroening, 2018). Recently, various studies have been conducted to improve localization effectiveness on SBFL fault localization techniques (N Neelofar, Naish, & Ramamohanarao, 2018; Yong Wang, Huang, Fang, & Li, 2018; X.-Y. Zhang, Zheng, & Cai, 2018). The studies have recorded improvements in terms of fault localization effectiveness.

2.2.2 Statistical-based Fault Localization Technique

Statistical-based fault localization techniques also exploit program run-time information (test case executions and their results) for locating faults. These techniques are predicate-based, unlike SBFL techniques that only utilize tests execution information and their results at the statement level. Predicates in programs are evaluated based on run-time information to assign a suspicious score for each predicate. The predicates will be ranked based on their suspicious scores for a debugger to find the faults. Therefore, Statistical-based fault localization techniques use predicate runtime execution data to understand program entities' correlation to failure. In a previous study by Liu et al., a statistical-based debugging algorithm named SOBER was proposed to rank suspicious predicates in a single run and also isolate faults in a program with multiple faults (Liu et al., 2006). SOBER classifies the effects of different faults and identifies predicates that are related to individual faults. These predicates explain the conditions and frequencies of fault occurrences and make it easier to prioritize debugging effort.

In addition, Wong et al. proposed a crosstab-based method for fault localization (E. Wong et al., 2008). A crosstab is constructed for each statement with two vertical categories (covered/not covered) and two horizontal categories (passed execution/failed execution). The researchers used a hypothesis test to provide a position of dependency/independency between the execution results and the coverage of each statement. The exact suspiciousness of each statement depends on the degree of association between its coverage and the execution results. Furthermore, Liblit et al. proposed a statistical debugging algorithm to isolate faults in a software program with instrumented predicates (Liblit et al., 2005). For each predicate, the algorithm computes the probability of that predicate being true that implies failure. Therefore, predicates that have a failure are identified. This creates a relationship between predicates and faults in

a program and the predicates are ranked based on their suspicious score. Predicates with high suspiciousness score are checked first and if a fault is found and fixed, the fault data is then removed. The process will be repeated to find the remaining faults until all predicates are examined.

You et al. proposed a statistical debugging approach by exploiting the statistical behavior of two closely connected predicates in a given program execution (You, Qin, & Zheng, 2012). For each test case, the approach constructs a weighted execution graph with predicates as vertices against change between the predicates as edges. For each edge in the graph, a suspicious score is computed to identify its fault relevant likelihood. Additionally, a novel probabilistic model for fault localization based on an important sampling of program statements was proposed in (Namin, 2015). By utilizing probability updates and sampling, the approach can help identify those statements that have a high likelihood of being faulty. The approach was found to be more sensitive to failed test cases than passed test cases.

2.2.3 Model-based Diagnosis Technique

Model-based diagnosis techniques (MBD) have good fault diagnosis accuracy and are successfully used for fault localization in the past decades (Abreu & van Gemund, 2009; Mayer, Abreu, Stumptner, & van Gemund, 2008; Wotawa, Nica, & Moraru, 2012). These techniques observe the conflict in behavior between the system model and its current state to locate faults in a software program. In other words, models are directly generated from a program that may contain faults. Therefore, the variance observed between the program executions and the expected results are utilized to identify the program components that may be the root cause for the observed misbehavior. The main

weakness of MBD techniques are their high computational complexity which limits the techniques' application to only programs with few hundred lines of code.

Mateis et al. proposed the use of a value-based model for Java programs that can handle imperative program execution. Logic-based languages such as first order logic are utilized to model a program behavior and a program structure are defined with dependency-based models (Mateis, Stumptner, & Wotawa, 2000). The study is extended by Mayer et al. to enable the dependency-based model to handle unstructured control flows such as exceptions, recursive method calls, and jump statements in Java programs (Mayer & Stumptner, 2002; Mayer, Stumptner, & Wotawa, 2003). Baah et al. proposed a model named probabilistic program dependence graph (PPDG) to model the internal behavior of a program. The model conducts a probabilistic analysis of a program behavior, specifically the behaviors that may be related to faults (Baah, Podgurski, & Harrold, 2010). The study by Wotawa et al. based on source code analysis, a dependency-based model was constructed from a given program to represent program structure and behavior (Wotawa, Stumptner, & Mayer, 2002). If a test case fails during program execution, the conflict between the failed test cases and program model will be used to find faulty candidates during fault localization. An assumption will be made on program statements whether they are faulty or not. This assumption will be revised until a full explanation of program failure is obtained. The main limitation of the study is the focus on loop-free programs. However, efforts were made to mitigate this limitation (Mayer & Stumptner, 2004).

Könighofer et al. proposed an automatic model-based debugging method with both fault localization and fault repair (Könighofer & Bloem, 2011). An incorrect program and its specification as a form of assertions will be an input into the proposed method. This information will be used to localize faults. Then a template-based approach will be used

for fault repair to confirm that repairs are readable. The result shows that in terms of handling incorrect assumption, the method can handle it in both single and multiple-fault programs. Abreu et al. used a model-based approach to localize multiple-fault candidates (Abreu, Zoeteweij, & van Gemund, 2008). In the process, De Kleer's intermittent fault model was used to explain software component behavior. Furthermore, the model together with passed/failed execution result of test cases was used to find observed failures in a program. The approach has performed relatively well on Tcas program of the Siemens test suite. Furthermore, Abreu et al. also proposed a reasoning approach named Zoltar-S for single fault localization and Zoltar-M for multiple fault localization (Abreu, Zoeteweij, & van Gemund, 2009a, 2011). Zoltar-M uses Bayesian probability theory to rank multiple fault candidates. The findings of the study showed that the approaches can outperform both statistical-based debugging and SBFL techniques.

In another study by Dean et al., an algorithm based on a linear programming model was utilized to help localize both single fault and multiple faults (Dean et al., 2009). Based on an empirical study on Siemens test suite and Space programs, the researchers concluded that the algorithm is better than some SBFL techniques like Ample coefficient, Tarantula coefficient, and Jaccard coefficient.

2.2.4 Program Slice-based Technique

Program slice-based technique extracts a subset of program statements that can affect the value of variables at the point where a fault is manifested. Irrelevant program parts are removed so that the resulting slice will be obtained. Slicing techniques can be static (M. Weiser, 1984) or dynamic (Agrawal & Horgan, 1990; B. Korel & J. Laski, 1988). Static slicing was first proposed in 1979 by (M. D. Weiser, 1979). The advantage of static slicing

is that it reduces the amount of code a developer needs to search to find faulty statements. The idea is, if a test case failed due to an incorrect variable value, the fault should be found in the static slice related to that variable value. This helps the developer to narrow down the fault searching space. However, having a reasonable slice size is crucial. Slice size has a great effect on the effectiveness of program slicing technique.

Lyle and Weiser proposed a new approach that constructs a program dice (the set difference of two groups of static slices) to aid in reducing the search space for potential locations of fault (Lyle, 1987). Static slicing techniques have been experimentally shown to be beneficial in locating program faults (Kusumoto et al., 2002), and it has been applied for fault localization in binary executable programs (Kiss, Jász, & Gyimóthy, 2005), and programs that are type-checked (Tip & Dinesh, 2001). The study by Binkley et al. showed that a typical slice size of a static slicing for a program can be one-third of the program under test (Binkley, Gold, & Harman, 2007). Realistically, it may not be useful to give a developer such a huge chunk of code to search for faulty statements. In order to address this issue, researchers proposed dynamic slicing (Agrawal & Horgan, 1990; B. Korel & J. Laski, 1988). Dynamic slicing relies on the information of the test suite which is the test coverage gathered through dynamic analysis to determine the parts of a program that affect the value of a particular program variable. In other words, dynamic slicing can identify the program statements that do affect a specific value of interest at a specific location, instead of possibly affecting such a value as with static slicing. Dynamic slicing technique is different from static slicing technique because of the utilization of tests execution data (i.e. input sequence, test cases). Many studies such as (Agrawal, DeMillo, & Spafford, 1993; Richard A DeMillo, Pan, & Spafford, 1996; Korel, 1988) have utilized the dynamic slicing concept in program debugging.

Critical slicing is proposed by DeMillo et al., it is fundamentally dynamic slicing but uses with mutation-based testing technique to further narrow down the slice size (Richard A DeMillo et al., 1996). To further narrow down the slice, Gupta et al. proposed a forward dynamic slicing (Gupta et al., 2005). Zhang et al. proposed a multiple points dynamic slicing technique which is the intersection of three types of dynamic slicing techniques (i.e. forward dynamic slice, backward dynamic slice, and bidirectional dynamic slice) (X. Zhang, Gupta, & Gupta, 2007). Wotawa combined dynamic slicing with model-based diagnosis to identify the root causes of failure. Based on the dynamic slices for faulty variables obtained, the researcher constructs hitting-sets that contain one statement from each dynamic slice. The suspiciousness of program statement will be calculated based on the number of hitting-sets that cover a statement (Wotawa, 2010). One of the limitations of dynamic slicing techniques is their inability to capture execution omission errors (Zhang et al., 2007).

A relevant slicing was proposed by Gyimóthy et al. to locate statements that are directly responsible for execution omission errors (Gyimóthy, Beszédes, & Forgács, 1999). Weeratunge et al. proposed the use of dual slicing which is the combination of dynamic slicing and trace differencing to identify the root causes of omission errors in concurrent programs (Weeratunge et al., 2010). Execution slicing is an alternative approach to static and dynamic slicing. Execution slicing uses test case executions to locate program faults which is identified to be easier to construct (Agrawal et al., 1995). Ju et al. proposed an approach to localize faults using a hybrid of full slice and execution slice (Ju et al., 2014). The approach first computes full slices of failed test cases and the execution slices of passed test cases. The information is then used to construct a hybrid spectrum by intersecting both full slices and execution slices to effectively localize program faults. Recently, a slice-based approach is proposed by Mao et al. to capture the influences of program entities (Mao et al., 2014). Their approach captures the influence

of a program entity on a set of test runs and statistical analysis was utilized to measure program entities suspiciousness of being faulty. The result shows a significant improvement in terms of effectiveness.

2.2.5 Machine Learning-based Fault Localization Technique

Machine learning-based techniques are regarded amongst the most effective fault localization techniques (Wong at al., 2016). They are robust, adaptive and can produce models based on data with minimal human interaction. Machine learning-based techniques have been used in different fields of computer science such as image and natural language processing, and cryptography (Browne & Ghidary, 2003; Kung, Kim, & Mukhopadhyay, 2015). These techniques use three-stage layers of operation which are input, hidden, and output layers to train data and provide a result of suspicious statements in software programs (W. E. Wong & Qi, 2009). In a previous study, Briand et al. proposed a C4.5 decision tree algorithm to aid in effective and efficient localization of faults (Briand, Labiche, & Liu, 2007). It is one of the early machine learning algorithms used in fault localization to partition failed test cases into different partitions in the presence of many faults. Furthermore, Wong et al. proposed two machine learning-based techniques for fault localization, fault localization based on BP (back-propagation) neural network (W. E. Wong & Qi, 2009) and fault localization based on RBF (radial basis function) neural network (Wong et al., 2012). These techniques have shown to be more effective than even most of the state-of-the-art SBFL techniques. However, these techniques have problems of paralysis and local minima.

In another study by Zheng and Wang, a fault localization based on Deep Neural Network (DNN) was proposed to tackle the problems of paralysis and local minima (W.

Zheng et al., 2016). DNN was found to be very effective in comparison to other machine learning-based techniques. The result of the study showed that when less than 10% of program statements are examined, DNN can identify 73.77 % of faults in all faulty versions, which is above most of the current state-of-the-art techniques.

2.3. Multiple Fault Localization

This section provides an overview of multiple fault localization. Jones et al. reported that the effectiveness of a fault localization technique declines on all faults as the number of faults increases. However, researchers also note that these results may be misleading and require further study (J. A. Jones et al., 2002). A later study by Jones et al. proposed the partitioning of failed tests caused by different faults to remove the noise caused by one fault inhibiting the localization of another fault (James A Jones et al., 2007).

Denmat et al. have made similar claims on the Tarantula coefficient (in extension to other similar SBFL techniques). The authors make a hypothesis requiring the independence of multiple faults where every failure is said to be caused by a single fault. However, when these hypotheses do not hold, the technique does not provide good fault localization results (Denmat, Ducassé, & Ridoux, 2005). In the study by Zheng et al., a specialized technique to localize faults in software programs containing multiple faults was proposed. Because in the existence of multiple faults, SBFL techniques cannot distinguish between fault infection and fault propagation that does not lead to failure (Zheng et al., 2006). In relation to SBFL techniques, Srivastav et al. indicated that the existence of multiple faults in a program prevents developers from effectively localizing a fault (Srivastav et al., 2010). In another study by Debroy and Wong, the researchers showed that wrong matching of failed test to fault may result in a poor fault localization

result (Debroy & Wong, 2009). Thus, programs with multiple faults as explained by existing studies suggest that it is difficult to match a failed test to its causative fault which in turn results in poor fault localization (DiGiuseppe & Jones, 2011b, 2015).

Drawing such conclusion is not unreasonable, as many empirical studies have been done to investigate the effects of having more than one fault in a program on localization effectiveness (DiGiuseppe & Jones, 2011b; Xue & Namin, 2013). Certainly, studies have shown that for a fault, the presence of other faults may impair the ability of SBFL techniques to properly localize them (James A Jones et al., 2007; J. A. Jones et al., 2002; Steimann & Bertschler, 2009). These studies conclude that SBFL performed poorly if, in a multi-fault program, it was unable to localize all faults effectively. This presumed that poor localization has led to the idea of localizing faults simultaneously which results in much literature in multiple fault localization. However, the main issue is interference between faults which causes the reduction of localization effectiveness in existing fault localization techniques.

2.3.1. Fault Localization Interference

Five investigative studies conducted to access the impact of fault interference on localization inferencing on programs with multiple faults and identify which type of interference is the most prevalent were discovered. Debroy and Wong investigated the occurrence of fault interference and further examined which form of interference occurs more often than another across all conditions (Debroy & Wong, 2009). The authors identified two forms of interference which are constructive and destructive interference. Constructive interference occurs when a test that passed in the presence of single fault failed in the presence of multiple faults. On the other hand, destructive interference occurs

when a test that failed in the presence of single fault passed in the presence of multiple faults. The result of the study shows that interference between faults do occur exponentially where the more faults exist in a program, the higher the frequency of faults interfering. The authors concluded that destructive interference is more common.

DiGiuseppe et al. conducted three studies on the effect of interaction of faults within a program (DiGiuseppe & Jones, 2011a, 2011b, 2015). Their studies indicated that the impact on localization effectiveness is real in the presence of multiple faults. Additionally, the authors concluded that even in the presence of many faults, at least one fault can be localized with good effectiveness. In another study by Xue and Namin, the researchers work on verifying the existence of fault interference phenomenon in object-oriented software programs (Xue & Namin, 2013). The result shows that fault interference occurs, however, its impact on performance is negligible where the effectiveness of localizing the first fault is not compromised.

Furthermore, two main approaches used in the localization of multiple faults are identified, namely OBA debugging approach and parallel debugging approach. In the next sections (Section 2.3.2 and Section 2.3.3), the studies that utilized these approaches are presented.

2.3.2. One-Bug-at-a-Time Debugging Approach (OBA)

OBA debugging approach is the process whereby a developer needs to localize a fault, fix it, and then re-test the program to find other faults in the software program under test. This process is performed iteratively until all the faults are found and fixed. Various fault localization techniques such as SBFL technique, statistical-based technique, and machine learning techniques have utilized OBA approach in localizing multiple faults

(Debroy & Wong, 2009; J. A. Jones & Harrold, 2005; J. A. Jones et al., 2002; W. E. Wong et al., 2014; W. E. Wong, Debroy, & Xu, 2012). Also, when utilizing this approach, additional effort is needed for the developer to find and fix the faults. In the process, more faults can also be created while also resulting in longer time-to-delivery of software programs (DiGiuseppe & Jones, 2015; Jeffrey et al., 2009). This section highlights studies that utilized the OBA debugging approach for localizing multiple faults.

Experimental results of most of the existing state-of-the-art fault localization techniques on multiple-fault programs show a decrease in their effectiveness due to OBA debugging approach utilization (DiGiuseppe & Jones, 2011b, 2015). For instance, an empirical study based on Tarantula (J. A. Jones et al., 2002) showed that its effectiveness drops because interference between faults hinders its performance and the time it takes to produce a failure-free program have increased because more debugging iterations are needed. It was also concluded in the same study that despite the loss of effectiveness, Tarantula was able to localize at least one fault effectively. In the attempt to address the problem of decreasing in effectiveness in the context of multiple faults. Abreu et al. presented an approach named BARINEL, which combines the best of SBFL techniques and MBD techniques (Abreu, Zoeteweij, & Van Gemund, 2009b). In this approach (BARINEL), a program was modeled with execution traces while Bayesian reasoning is used to deduce multiple fault candidates. The approach showed some promise but was found to be more effective in the context of a single fault.

In a previous work by Gong et al., the authors proposed a mechanism to improve the localization efficiency of single-fault localizers by providing a stopping criterion in the first debugging iteration to allow developers to locate more than one fault (Gong et al., 2012). However, the approach was not tested and validated. Furthermore, a multiple fault localization method was proposed based on Simulink model (Liu et al., 2016). The

approach used supervised learning technique named decision tree to cluster failed executions that were likely to have been caused by a single fault. In this process, a rank list based on a statistical debugging technique is generated and developers can use this list to find a fault, fix it, and re-test the Simulink model to localize the remaining faults. Although this approach uses failure clustering method as the basis of classifying failures, it is still based on the OBA debugging approach as indicated in the study.

In another study, Lee et al. proposed a weighting technique to improve the effectiveness of SBFL techniques (Lee, Kim, & Lee, 2016). A weighting value is assigned to test cases that are caused by both single fault and multiple faults. This weighting is primarily done by utilizing information extracted from failed test cases that are caused by multiple faults. The study concluded that weighting failed test cases caused by multiple faults improve the effectiveness of fault localization techniques. Furthermore, Wong et al. proposed a modified form of *Kulczynski* similarity coefficient named DStar (W. E. Wong et al., 2014). Using this coefficient, the higher the computed DStar value, the more effective the technique. The coefficient was tested on multiple-fault programs using the OBA debugging approach and the findings revealed that the technique can localize at least a single fault with high effectiveness.

Wang et al. proposed a novel fault localization approach based on disparities of dynamic invariants, named FDDI (X. Wang & Liu, 2016). FDDI selects a highly-suspected function and then applies invariant detection tools to this function separately. Variables that are not in a set of passed/failed test cases indicated by using these tools are picked by FDDI for further analysis. However, in the context of multiple faults, the authors considered failed test cases that execute all faulty statements. Therefore, using the OBA debugging approach, the researchers neutralized single fault each at each debugging iteration to produce a failure-free program. In addition, Xu et al. proposed a

fault localization framework that reduces the noise between faults in the presence of multiple faults (Xu et al., 2013). The framework uses a chain of key basic blocks of a program and a noise reduction method to improve similarity coefficient metrics.

Likewise, Zhao et al. proposed a general framework to mitigate the effect of execution similarity and improve the effectiveness of SBFL techniques (Zhao et al., 2013). Using the existing SBFL formulae, the approach can reduce the impact of both execution similarity and improve fault localization effectiveness. In the context of multiple faults, the OBA debugging approach was utilized. To reduce the noise in fault-failure correlation when localizing faults caused by either coincidental correctness (Masri & Assi, 2014) or fault interference (Xue & Namin, 2013), another study by Xu et al. proposed a framework to address this issue. The study shows that the noise reduction framework improves the effectiveness of SBFL techniques when applied to multiple-fault programs (Xu et al., 2013).

In addition, a crosstab-based statistical approach was proposed by Wong et al. to localize faults using the OBA debugging approach (W. E. Wong, Debroy, & Xu, 2012). The experimental results showed that the OBA debugging approach is not as effective. Furthermore, a weighting technique was proposed by Neelofar et al. using both dynamic program analysis and static program analysis to categorize program statements and rank them based on their likelihood of containing faults. The technique was tested on both single-fault programs and multiple-fault programs. Their proposed technique improves the performance of various fault localization metrics up to 20% on single-fault datasets and up to 42% on multiple-fault datasets (Neelofar Neelofar et al., 2017). A diversity maximization speedup (DMS) strategy was proposed by Xia et al. to aid developers in the test case selection during fault localization to also reduce associated costs (Xia et al., 2016). This strategy also helps in targeting critical test cases that are needed to speed up

the localization process. The result of the study shows that DMS can aid the existing fault localization techniques in reducing debugging cost of locating multiple faults.

A hybrid method was proposed named Stat-slice to locate faults in programs with a large number of faults using the OBA debugging approach (Parsa, Vahidi-Asl, & Zareie, 2016). The result shows that the method has considerably reduced fault localization effort. Also, bounded debugging via multiple predicate switching (BMPS) technique based on the OBA debugging approach was proposed (A. Liu, Li, & Luo, 2015), and experimentation revealed that multiple faults were localized. Xiaobo et al. conducted an empirical study to explore failure behavior of multiple faults in a program through empirical investigation on real-life systems (Chinese Railway System) (Xiaobo, Bin, & Jianxing, 2017). The study showed that unpredictable failure caused by multiple faults is mainly accounted by the interaction of dominant faults during program execution.

Sun et al. proposed a novel approach named the concept lattice of program spectrum for effective multiple fault localization (CLPS-MFL) (Sun, Li, & Wen, 2013). The approach uses formal concept analysis to convert a program spectra into a concept lattice and uses three strategies to find failure root causes. However, the approach needs further improvement in the context of multiple faults. Generally, in an effort by researchers to localize multiple faults simultaneously in a single debugging iteration, MBD techniques have been used recently (Lamraoui & Nakajima, 2016). However, MBD techniques have limitations due to their computational complexity. Therefore, their use on large programs is limited and the results cannot be fully generalized.

2.3.3. Parallel Debugging Approach

Parallelization or parallel debugging approach is basically dividing the debugging task into small units so as to allow multiple developers to work on different units (James A Jones et al., 2007). This approach is utilized when a program has multiple faults, mainly to facilitate the debugging process and reduce software time-to-delivery. Failed test cases are clustered and each cluster is combined with all the available passed test cases to create a single *fault-focused* cluster, with the assumption that each *fault-focused* cluster targets a single fault. The *fault-focused* clusters composed of both failed and passed test cases will be given to separate developers to debug in parallel. This debugging approach is different from the OBA debugging approach because a developer does not have to neutralize the faults in many debugging iterations.

Jones et al. introduced the idea of debugging in parallel by clustering the failed test cases and combining each cluster with all available passed test cases to form a *fault-focused* cluster (James A Jones et al., 2007). These *fault-focused* clusters are then given to developers to debug the faults in parallel. However, the same study presumed that each *fault-focused* cluster represents a single fault. A recent work in (Högerle et al., 2014) concluded that the assumption does not seem realistic. One of the issues with parallelization is that, a single developer can finish a debugging task while other developers are still debugging. Consequently, this can create more faults to the program as fixes given to the first developer who finishes debugging early, will probably affect the debugging effort of other developers.

In some studies, failed test cases are clustered based on their execution profile similarity. The clustering stops when the two clusters to be merged further seem to target different faults (James A Jones et al., 2007; Yu et al., 2008). This clustering technique for isolating faults is claimed to be inappropriate and problematic (R. Gao & Wong, 2017).

In another study by Srivastav et al., a technique was proposed to calculate the complexity of program slices before distributing the work to the respective debuggers (Srivastav et al., 2010). After clustering, the technique calculates the slice weight to determine the effort needed to debug each slice. However, the approach needs to be empirically proven to justify its usability.

Jeffrey et al. proposed a fault localization method based on value replacement to efficiently localize multiple faults. The technique reduces the total time required to locate multiple faults on the order of minutes. Initially, finding and fixing the faults based on OBA debugging approach was considered, before the authors eventually concluded that it would be costly and that such approach also increases time-to-delivery of the software program (Jeffrey et al., 2009). As a result, an iterative process was considered, where the technique can iteratively compute a ranking list of program statements with the aim of each ranking list to guide the developer towards faults as quickly as possible. The method understands potential faulty statements based on the occurrence of interesting value mapping pair (IVMP) (Jeffrey et al., 2009). It was identified that IVMP normally occurs at faulty statements. It can also occur in statements that are directly related to faulty statements through a dependency edge. The approach performs localization on individual execution iterations to find and fix faults, which have similarities with the approach by Jones et al. (James A Jones et al., 2007). Moreover, it is not guaranteed that each iteration belongs to a single fault. One of the problems with this approach is the high computational requirements. Even though some faults can be identified in a matter of minutes, others can take hours.

In another study by Wei and Han, a parameter-based combination approach (PBC) was proposed to aid in the efficient localization of multiple faults (Wei & Han, 2013). In the study, a bisection method was utilized for clustering failed test cases to create *fault-*

focused clusters while crosstab-based fault localization technique was used for fault localization. The researchers conclude that PBC performs better than Tarantula (OBA debugging approach). In addition, two other studies proposed the use of parallel debugging approach to improve localization effectiveness on multiple-fault programs (Briand et al., 2007; W. E. Wong, Debroy, Golden, et al., 2012). The result showed significant improvement in contrast to adopting the OBA debugging approach.

In a recent study by (Lamraoui & Nakajima, 2016), a formula-based approach was proposed consisting of a full flow-sensitive trace formula to localize faults in programs containing multiple faults. The approach combines satisfiability-based (SAT) formula verification techniques and model-based diagnosis theory. It was able to localize the root causes of multiple faults in a program. However, this approach was examined in a relatively small program of the Siemens test suite (Tcas), and therefore, the results cannot be fully generalized. Also, the approach's exclusive utilization of failed test cases alone for localization might be an issue for a large program with several faults where many statements containing faults can be executed by passed test cases which are more common in multiple-fault scenarios. Sun et al. proposed an iterative process for selecting test cases for effective fault localization (Sun et al., 2016). This approach works based on the concept lattice of program spectrum method, and in order to localize multiple faults, program statements are classified into three parts namely, dangerous, sensitive, and safe statements. By doing so, developers start by checking statements that are categorized as dangerous first due to their high probability of containing faults, followed by the rest of the classified statements.

To simplify debugging efforts in statistical debugging technique, Liblit et al. proposed an algorithm to isolate faults in programs with multiple faults (Liblit et al., 2005). The algorithm is tasked to identify predicates that are correlated with specific

singular faults and isolate them in order to prioritize debugging effort to localize faults simultaneously. A more recent approach proposed is called Hierarchy-Debug (Parsa, Vahidi-Asl, & Asadi-Aghbolaghi, 2014), which aims to localize latent bugs. In this approach, a hierarchical clustering algorithm is applied to cluster predicates to support scalability in localizing multiple bugs. The results showed that the approach can aid developers in grouping predicates caused by multiple bugs. A recent study by Gao and Wong proposed a novel approach for localizing multiple faults in parallel (R. Gao & Wong, 2017). The authors proposed an improved *k*-medoids clustering algorithm to aid in the effective identification of the relationship between failed test cases and their corresponding faults. The study concludes that their proposed approach performs better in terms of efficiency and effectiveness in comparison to other debugging approaches.

2.4. Complex Network Theory

Complex network is capable of simulating complex data behavior to understand and identify important components (Dorogovtsev & Mendes, 2003). For the past few decades, researchers in diverse fields of studies have given a lot of attention to complex network theory due to its robustness and adaptiveness in solving complex problems (Albert & Barabási, 2002; Bornholdt & Schuster, 2006; Strogatz, 2001). Particularly, physicists have shown a lot of interests in complex network to provide a detailed explanation of various system topologies such as social networks, communication systems, World Wide Web (WWW), community structures, epidemic spreading and more. Complex network has a lot of advantages over many models used in the study of complex data. It has the ability to learn highly complex data and structures of a system. The working principle of complex network is flexible and robust because the complex relationships of system components can be understood at a macro level. Most importantly, complex network has

a strong mathematical background. These advantages have led complex network in becoming an important tool for understanding system complexity. It has been successfully applied in the scientific research areas such as neurology (Strogatz, 2001), biology (Dorogovtsev & Mendes, 2003), physics (Albert & Barabási, 2002), social network (L. C. Freeman, 1978), and software engineering (Ma, He, & Du, 2005; Myers, 2003).

Albert and Barabási conducted a study to understand networks and the complexity of World Wide Web (WWW) from a physicist point of view (Albert & Barabási, 2002). The authors try to understand the basic principles of network structural organization and evolution. The study by (Strogatz, 2001) evaluate some of the most basic issues in networks in the area of neurobiology from the perspective of nonlinear dynamics. The outcome of the study shows that there are issues about the nonlinear dynamics of systems coupled according to small-world, scale-free or generalized random connective network. Furthermore, in order to capture the structural characteristics of an object-oriented software system, a study by Chong and Lee proposed an approach to represent software systems using weighted complex network (Chong & Lee, 2015). Based on the complexities of classes and their dependencies (methods), nodes and edges are modeled. Graph theory metrics were used on the modeled network. The result showed that their approach can help in identifying software components that violate software design principles. Ma et al. proposed a qualitative measure based on software structure entropy that measures the amount of uncertainty of the structural information. The researchers further measured the influence of interactions between the components of software systems and their topologies/structures (Ma et al., 2005). The study by Myers (Myers, 2003) examined software collaboration graphs of several open-source systems, and the findings show that software graphs indicate small-world and scale-free network characteristics which are identical to those identified in other systems (sociological,

biological, neurological and so on). Zhu et al. proposed a fault localization method based on software network centrality measures (SNCM) to improve localization effectiveness on single fault subject programs. The result shows that their method is useful in identifying the location of faults (Zhu et al., 2011).

Complex network or sometimes called graph theory has two basic types of graphs which are, directed graph and undirected graph. A graph is a group of nodes connected together via edges that may or may not be weighted. There are two types of node-to-node relationships, symmetric and asymmetric. Nodes relationships are symmetric if the graph is modeled as an undirected graph, while the nodes relationships are asymmetric if the graph is modeled as a directed graph (Bullmore & Sporns, 2009). Basically, a graph provides an abstract representation of the modeled data, for instance, social network, complex data or software system components and their interactions, which is often regarded as a real-world network because it simulates real-world behavior of a system.

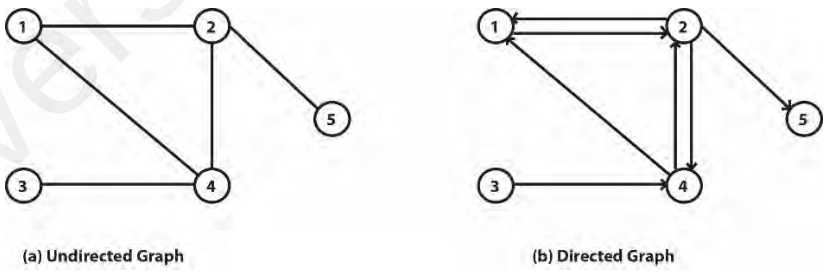


Figure 2.2: Directed graph and undirected graph

Graph (a) in Figure 2.2 is referred to as undirected graph because the nodes' interactive relationships are not shown in detail. On the other hand, graph (b) is referred to as directed graph; it shows a detailed interaction between nodes. Node 1 and node 2 in graph (b) share information in a bi-directional manner with two edges between the nodes.

The same is applied to node 2 and node 4 in graph (b). Real-world networks show essential topological structures, patterns, and behaviors of a system. Existing studies have found that real-world networks possess some unique characteristics and behavior such as the scale-free and small-world properties, which are not found in random networks (Myers, 2003; Šubelj & Bajec, 2012). A directed graph is more suitable when applied in object-oriented software systems because it can capture the semantic relationships between software components. In this thesis, the detailed relationships between components as the case in object-oriented systems (Ma et al., 2005) are not of interest. The graph is modeled as an undirected graph. To build a software-based complex network, a developer needs to have a valid input regarding the structural behavior of the software such as software components and their relationships. In this thesis, the input values will be program spectra and the execution results (passed/failed) of a program. The program statements will be modeled as nodes and the tests execution between them will be modeled as edges. Hence, the term program statement and node are used interchangeably in this work.

In this thesis, complex network is used for fault localization for the following reasons. Firstly, complex network theory is proven to be largely applicable theory and it has been effectively used to solve several problems in research areas such as physics (Albert & Barabási, 2002), biology (Dorogovtsev & Mendes, 2003), social network (L. Freeman, 2004), and software engineering (Chong & Lee, 2015). Secondly, complex network has the ability to help researchers understand complex systems. For instance, it can be used to identify important nodes and their correlation with faults in software programs (Zhu et al., 2011). Lastly, in the context of fault localization, complex network theory has the ability to aid in the identification and localization of faulty program statements and statements that are related to failure.

2.5. Research Gap

Based on the research literature presented in this chapter, the key observation is that in programs with multiple faults, the localization effectiveness of the existing fault localization techniques reduces. The more faults a program contains, the more difficult it is to localize all the faults simultaneously (DiGiuseppe & Jones, 2015). Next, looking at all the studies utilizing OBA and parallel debugging approaches, none of them localize faults in a single diagnosis rank list. In other words, no study localize faults in a single debugging iteration. An earlier study by Gong et al. has made a suggestion on localizing more than one fault in the first debugging iteration, in extension, debugging all the faults simultaneously (Gong et al., 2012). However, their proposal has not been implemented or validated. A recent study by Zheng et al. has also put an effort to localize faults simultaneously by proposing a Fast Software Multi-Fault Localization Framework based on Genetic Algorithms (Y. Zheng et al., 2018). In view of the research gap, to improve localization effectiveness and reduce the overall software time-to-delivery, this thesis aims to develop a fault localization technique that will localize multiple faults in a single debugging iteration.

Furthermore, clustering is the most important component of parallel debugging approach, particularly on how to obtain a good clustering on failed test cases that target single faults. Many clustering algorithms have been used in the literature such as *k*-means clustering algorithm, hierarchical clustering algorithm, and *k*-medoids clustering algorithm (R. Gao & Wong, 2017; Högerle et al., 2014; James A Jones et al., 2007; Steimann & Frenkel, 2012; Yabin Wang et al., 2014). Clustering algorithms that group failed test cases based on their execution profile similarity with distance metrics such as Euclidean distance, Jaccard distance, Hamming distance and so forth, are claimed to be problematic and inappropriate when utilized in a parallel debugging approach in the

context of multiple faults (R. Gao & Wong, 2017). Yet, other studies that grouped failed tests based on their execution profile similarity with the above-mentioned distance metrics in parallel debugging approach recorded good localization results (Huang et al., 2013). However, there are no studies that investigate the claimed problematic parallel debugging approach that groups failed test cases based on their execution profile similarity with a distance metric such as Euclidian distance in localizing multiple faults in terms of localization effectiveness. As far as the knowledge gained from the literature reviewed in this domain, apart from the study by Zhu et al. that makes use of complex network theory to localize faults on single-fault programs, complex network theory has not been used for fault localization (Zhu et al., 2011).

In this thesis, an investigative study of the claimed problematic parallel debugging approach that makes use of *k*-means clustering algorithm to group failed tests execution based on their execution profile similarity with Euclidean distance metric is conducted. The claimed problematic parallel debugging approach is also compared with OBA debugging approach and MSeer parallel debugging approach (R. Gao & Wong, 2017) in terms of localization effectiveness. Furthermore, in the quest of improving software quality and effectively localizing multiple faults simultaneously, this thesis proposes two new fault localization techniques based on complex network theory that will aid in localizing both single and multiple faults effectively in a single diagnosis rank list. Furthermore, a new approach for isolating faults into different clusters is proposed to aid in the simultaneous localization of multiple faults in parallel. The new approach does not perform clustering on tests execution, rather it performs clustering on program statements in a complex network that is modeled based on the tests execution profile. Therefore, the *due-to* relationship between program statements is measured based on statements' edge-betweenness distance to create *fault-focused* communities (clusters) instead of between failed test cases as done by the previous works.

2.6. Chapter Summary

This chapter has presented the literature on software fault localization. The chapter started by highlighting some of the basic terminologies that are vital in the study of faults to failure relationship between program entities in software programs. The chapter has presented some of the most prominent software fault localization techniques in the research domain. The chapter further discussed multiple fault localization, fault interference, and the two main debugging approaches used in debugging multiple faults. Studies utilizing these approaches to localize multiple faults are highlighted in detail. The chapter also gives a basic background literature on complex network theory and some related works on its application in various research domains including software engineering. The justification of using complex network theory for fault localization was also given. Finally, the chapter concluded by highlighting the gaps in research and the need for this research. In conclusion, this chapter views the strength and weaknesses of the previous works based on the fault localization techniques and the debugging approaches utilized and their performance in the context of multiple faults. In the next chapter, the research methodology will be described and discussed in detail.

CHAPTER 3: RESEARCH METHODOLOGY

In this chapter, the general methodology of the study is discussed. The methodology of the investigative study of the claimed problematic parallel debugging approach that makes use of *k*-means clustering algorithm with Euclidean distance metric in localizing multiple faults is highlighted. The two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), are presented in detail. Lastly, the new community-based fault isolation approach is also presented in detail.

3.1. Research Process

Firstly, the whole research process is divided into four stages as shown in Figure 3.1. The first stage is literature review which focuses on the existing software fault localization techniques to identify their contributions and limitations. In this stage, a study of the existing state-of-the-art fault localization techniques is done to identify the issues and challenges the existing techniques faced when localizing multiple faults (Zakari et al., 2018). Next, the studies were categorised based on the fault localization techniques and debugging approaches they utilized to localize multiple faults. Two multiple-fault debugging approaches utilized by researchers to localize multiple faults, namely OBA debugging approach and parallel debugging approach, were identified and studies utilizing these approaches were discussed. Furthermore, the basic background literature on complex network theory and some related works on its application in various research domains including software engineering were given.

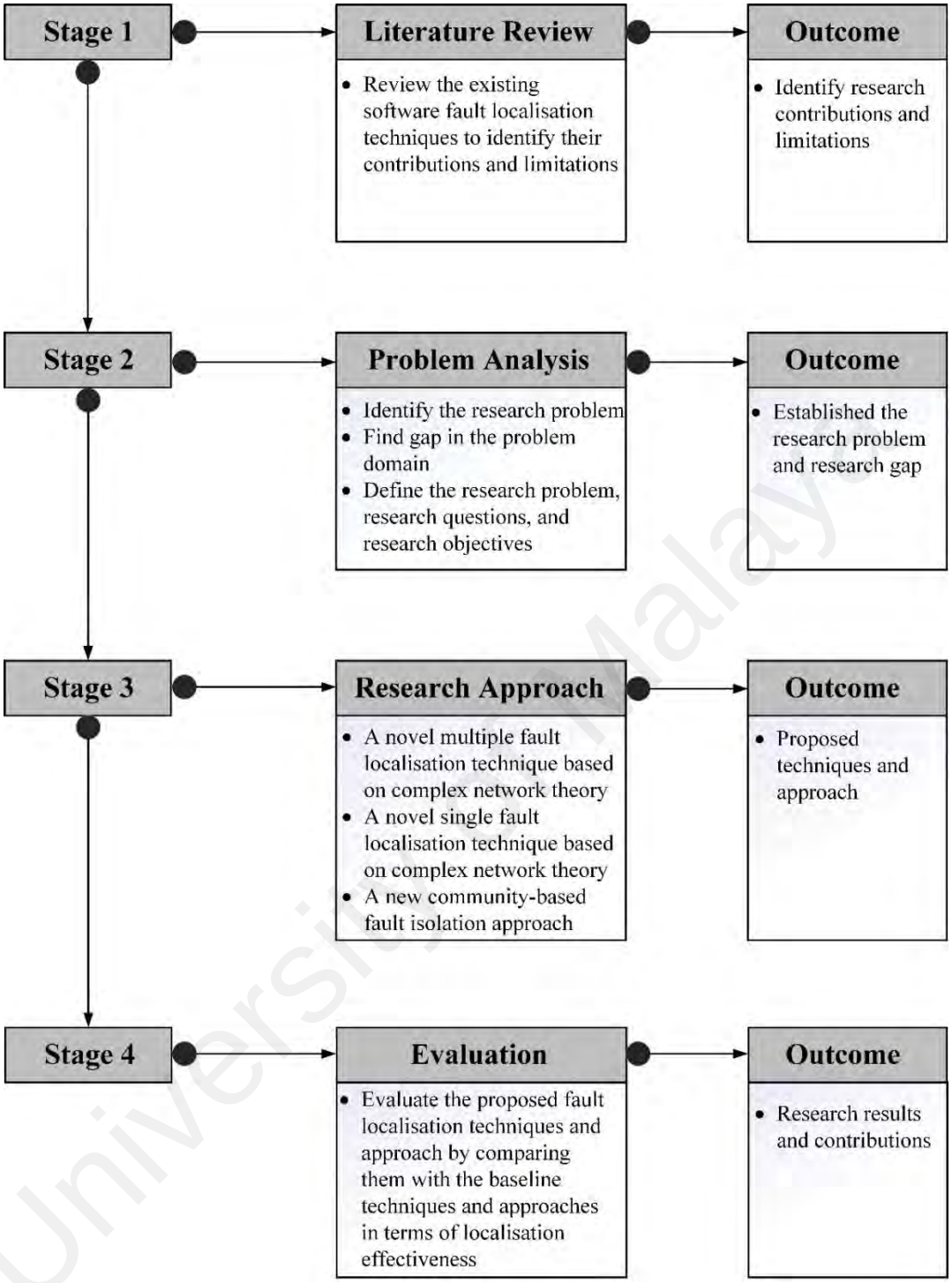


Figure 3.1: Process flow of the research activities

Secondly, the next stage is the problem analysis. The problem statement was formulated based on the analyzed literature. It was observed that most of the existing studies show that the existence of multiple faults in a program reduces the effectiveness of the existing fault localization techniques (DiGiuseppe & Jones, 2015; Xue & Namin,

2013). Thus, the techniques cannot localize faults simultaneously in a single diagnosis rank list. Moreover, parallel debugging approach is also claimed to be problematic particularly when failed tests are clustered based on the similarity of their execution profile with distance metrics such as Euclidean distance, Jaccard distance, and Hamming distance (R. Gao & Wong, 2017; C. Liu et al., 2008). To substantiate this claim, an investigative study of the claimed problematic parallel debugging approach that makes use of *k*-means clustering algorithm (that groups failed test cases based on their execution profile similarity) with Euclidean distance metric is conducted. The details of the investigative study is presented in Section 3.2. Furthermore, a set of research questions and research objectives are formulated to solve the identified research problem as highlighted in Chapter one, Section 1.4 and Section 1.5.

Thirdly, the next stage is the research approach. This stage focuses on proposing solutions to solve the research problem and fulfill the research objectives. First of all, in order to localize multiple faults simultaneously in a single diagnosis rank list with good effectiveness, a novel fault localization technique based on complex network theory named multiple fault localization based on complex network theory (FLCN-M) is proposed. This technique is specifically built to improve localization effectiveness in multiple-fault programs and to aid developers in localizing multiple faults simultaneously in a single diagnosis rank list. The details of the proposed technique is presented in Section 3.3. In addition, a novel technique named single fault localization based on complex network theory (FLCN-S) is proposed to localizing faults on single-fault programs. The details of the proposed technique is also presented in Section 3.4. Furthermore, for FLCN-M fault localization technique, in the case where a developer has checked 70% of the program statements and cannot fully localize all the multiple faults in a single diagnosis rank list, instead of resorting to using the OBA debugging approach, a newly proposed community-based fault isolation approach that makes use of a divisive

network community clustering algorithm is applied to aid in the isolation and localization of multiple faults simultaneously in parallel. The details of the proposed approach is presented in Section 3.5.

In the last stage, the proposed techniques and approach were evaluated on several single-fault and multiple-fault subject programs in comparison with various baseline fault localization techniques and approaches in terms of localization effectiveness. For the evaluation of the proposed FLCN-M fault localization technique, a newly generic evaluation metric named incremental developer expense (IDE) is proposed to aid in accessing developer expense in localizing multiple faults simultaneously in a single diagnosis rank list.

3.2. Investigative Study of the Claimed Problematic Parallel Debugging Approach

This section presents the methodology of the investigative study on the claimed problematic parallel debugging approach with the clustering algorithm, distance metric, and fault localization techniques utilized. The work on the claimed problematic parallel debugging approach is primarily to investigate its usefulness in terms of localization effectiveness on multiple-fault programs in comparison with two other debugging approaches.

3.2.1. Parallel Debugging

The basic idea for parallelizing debugging activity was introduced by Jones et al (James A Jones et al., 2007). The goal is to isolate different faults in separate special test suites for multiple developers to debug in parallel, ideally to reduce debugging cost. The

motivation of utilizing parallel debugging approach comes from the parallelization of computation on multi-processor computers (Despain & Patterson, 1978). This type of approach divides a task into multiple subtasks that are processed on multiple processors simultaneously. Hence, because of the better utilization of the processors, the tasks complete faster when utilizing parallel approach rather than a sequential approach.

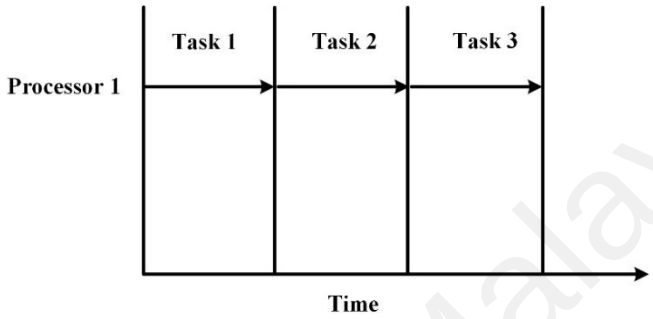


Figure 3.2: Sequential processing of tasks

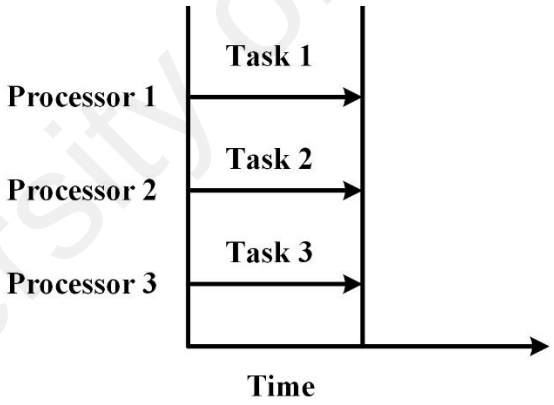


Figure 3.3: Parallel processing of tasks

Figure 3.2 and Figure 3.3 illustrate the sequential and parallel computation of tasks. The horizontal axis represents the cost of the tasks and the vertical axis shows the processors (developers in this work context) that are attached to each task. In the fault localization context, the former (Figure 3.2) represents the OBA debugging approach while the latter (Figure 3.3) represents the parallel debugging approach. It is obvious that the latter utilizes less time in completing the debugging task. Moreover, debugging task completion can be

measured as the time and effort it takes to locate the faults responsible for the failure, and debugging task parallelization can also be measured as the number of developers that can debug the program.

	Test Cases											
	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10		
mid ( ) { int x,y,z,m;	3,3,5	1,2,3	3,2,2	5,5,5	1,1,4	5,3,4	3,2,1	5,4,2	2,1,3	5,2,6	suspiciousness	rank
1: read ("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	●	●	●	●	0.50	9
2: m = z;	●	●	●	●	●	●	●	●	●	●	0.50	9
3: if (y < z)	●	●	●	●	●	●	●	●	●	●	0.50	9
4: if (x < y)	●	●			●	●			●	●	0.43	10
5: m = z; // fault1. correct: m = y		●	●	●	●	●			●		0.65	8
6: else if (x < z)	●				●	●			●	●	0.50	9
7: m = x;	●				●				●	●	0.60	4
8: else			●	●			●	●			0.60	4
9: if (x > y)			●	●			●	●			0.60	4
10: m = z; // fault2. correct: m = y			●				●	●			0.75	1
11: else if (x > z)				●							0.00	13
12: m = x;											0.00	13
13: print ("Middle number is: ", m);	●	●	●	●	●	●	●	●	●	●	0.50	9
} Pass/Fail Status	P	P	P	P	P	P	F	F	F	F		

Figure 3.4: Program with all test cases

Given a program P with a test suite *T* in Figure 3.4. Test suite *T* is composed of 10 test cases (t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}) with 13 program statements where statement 5 and statement 10 are both faulty. A statement execution labeled with "●" signifies that the statement is executed by the test case in that test run, and empty otherwise. Test cases (t_7, t_8, t_9, t_{10}) are failed test cases while (t_1, t_2, t_3, t_4, t_5, t_6) are passed test cases. With two faults in P, locating both faults simultaneously will take more time using the OBA debugging approach as it requires retesting the program iteratively to find more faults which will result in the increase of software time-to-delivery.

Hence, to localize all the faults in a minimum amount of time with minimal developer expense, a parallel debugging approach will be utilized. *Fault-focused* clusters

will be created by automatically clustering the failed test cases into subsets that has similar tests execution profile. Each of these *fault-focused* clusters will be combined with all available passed test cases to create a special test suite. These special test suites will be given to individual developers to debug the faults in parallel. With these special test suites, a fault localization technique will be used to automatically find the faults. In this study, *k*-means clustering algorithm will group failed tests execution based on their execution profile similarity with Euclidean distance metric to measure the distance between two failed tests.

Considering Figure 3.4, based on the algorithm (*k*-means), failed tests execution (t_9, t_{10}) and (t_7, t_8) will be partitioned into different clusters that target different faults. Each of these clusters will be combined with all the passed tests execution to create a special test suite that is presumed to target a single fault. Clustering failed tests based on their execution profile similarity was also used in previous fault localization studies (Högerle et al., 2014; Huang et al., 2013; C. Liu et al., 2008). However, as suggested by Liu et al. (C. Liu et al., 2008) and Gao et al. (R. Gao & Wong, 2017), this representation is considered to be problematic because of the unpredictability of fault manifestation (faults can be triggered in many ways). These studies also argued that this representation resulted in poor fault localization results due to a poor clustering method utilized. Therefore, this approach will be considered for the investigative study. The next section highlights the *k*-means clustering algorithm used in the investigative study.

3.2.2. K-means Clustering

Clustering is the partitioning of a collection of objects into clusters (*k*) that have similar behavior (Witten & Frank, 1999). Objects partitioned in the same cluster are similar and are not similar to other objects in separate clusters.

In general, clustering techniques can be classified into two categories, which are soft clustering techniques and hard clustering techniques. In the former, an object can be a member of two or more clusters, and in the latter, an object can only be a member of one cluster. In this study, *k*-means clustering algorithm is used for clustering failed tests execution (Hartigan & Wong, 1979; Yabin Wang et al., 2014). *k*-means clustering algorithm is one of the most popular hard clustering techniques, and it was utilized by few studies in the domain of software fault localization (Huang et al., 2013; Yabin Wang et al., 2014). Before *k*-means is applied, the number of clusters, *k*, has to be estimated based on the total number of failed test cases N_t. Furthermore, based on the estimated number of clusters, the cluster centers will be randomly selected for each cluster where each failed test is assigned to the nearest cluster center based on the Euclidean distance between failed test cases. Lastly, the cluster centers will be recalculated based on the existing clustering results. Thus, in each clustering iteration, new cluster centers are produced, and the process continues until test cases in each cluster no longer move. In *k*-means clustering, clusters are produced so as to minimize Equation 3.1 as stated in (Yabin Wang et al., 2014).

$$s = \sum_{i=1}^{k} \sum_{t_j \in s}^{Q} ||t_j - v_i||^2 \qquad (3.1)$$

where t_j represents a test case, v_i represents a cluster center, s is the *ith* cluster, k represents the number of clusters, Q represents the number of test cases in the *ith* cluster, and $\|t_j - v_i\|^2$ is the square distance between t_j and v_i. Therefore the clustering of failed test cases into k clusters include the following steps:

Step 1: Before *k*-means is applied, the number of clusters, *k*, will be estimated as $\sqrt{N_t/2}$ based on the work in (Bibby, Kent, & Mardia, 1979) and (Yabin Wang et al.,

2014) where N_t represents the total number of failed test cases. Although a recent study by Gao et al. (R. Gao & Wong, 2017) argues that the above cluster estimation is problematic because there is no clear correlation between the number of failed test cases and the number of faults in a given program. However, the study in (Yabin Wang et al., 2014) shows this approach can be useful in estimating the number of clusters that target single faults. Hence, this study will aid in investigating this issue on whether using this clustering algorithm with this cluster estimation will result in effective fault localization results.

Step 2: knowing the number of clusters from step 1, the number of cluster centers v will be randomly assigned corresponding to the number of clusters where v can be represented as $v = \{v_1, v_2, v_3, \ldots, v_n\}$. Suppose there are two clusters ($k = 2$), v_1 and v_2 will represent the cluster centers which is assigned to the mean point in a group of failed tests.

Step 3: To calculate the Euclidean distance between a test case t to all cluster centers, Equation 3.2 will be computed.

$$d(t, v) = \sqrt{\sum_{j=1}^{Q} (t_j - v)^2} \qquad (3.2)$$

In this work, the execution profile of a test case t is represented as a numeric vector $t = \{st_1, st_2, st_3, \ldots, st_n\}$ where each t is executed by n number of statements which can be represented as $m = \{m_1, m_2, m_3, \ldots, m_n\}$. Therefore, as shown in Equation 3.3, if a statement execution is represented as $m_i = 0$, it indicates that the statement is not executed by t, while if it is represented as $m_i = 1$, it indicates that the statement is executed by t.

$$m_i = \begin{cases} 1, & executed \\ 0, & not\ executed \end{cases} \qquad (3.3)$$

A cluster center v is also a numeric vector holding the same dimension as t. Therefore, v can be represented as a numeric vector $v = \{sv_1, sv_2, sv_3, \ldots, sv_n\}$. For clarification, refering to Table 3.1 where t can be represented as $t = \{st_1, st_2, st_3, \ldots, st_n\}$, the numeric vector values for test case t_1 and t_2 is represented as $t_1 = \{1, 1, 1, 0, 0, 0, 1\}$ and $t_2 = \{1, 0, 0, 1, 1, 0, 1\}$.

Next, the Euclidean distance between numeric vector values of t and numeric vector values of v will be computed to know the distance between t and v as $d(t, v)$. Having two cluster centers v_1 and v_2, to calculate the Euclidean distance for the first cluster center v_1 against all test cases, Equation 3.4 will be computed.

$$d(t, v_1) = \sqrt{(st_1 - sv_1)^2 + (st_2 - sv_2)^2 + (st_3 - sv_3)^2 +, \ldots, (st_n - sv_n)^2} \quad (3.4)$$

Hence, to calculate the Euclidean distance for the second cluster center v_2 to all test cases, Equation 3.5 will be computed.

$$d(t, v_2) = \sqrt{(st_1 - sv_1)^2 + (st_2 - sv_2)^2 + (st_3 - sv_3)^2 +, \ldots, (st_n - sv_n)^2} \quad (3.5)$$

Lastly, to assign t to the nearest cluster center whose distance from the cluster center is minimum to all cluster centers, Equation 3.6 will be computed, where s represents the *ith* cluster. Therefore, the same will be done to all the test cases.

$$s = \left\{t: ||t - v_1||^2 \leq ||t - v_2||^2 \leq, \ldots, ||t - v_n||^2\right\} \quad (3.6)$$

Having the total distance calculated by Equation 3.4 and Equation 3.5. Therefore, for each t, it is assigned to only one s that is nearest to it.

Step 4: To calculate and update the mean point of each cluster center v based on the existing clustering results, Equation 3.7 will be computed where $|s|$ indicates the number of failed test cases in s. Therefore, the clustering result for each of the test cases in s is

indicated as $d(t_j)$ where they will be added up and divided by the total number of failed test cases in s to update the mean point of each of the cluster centers for the next clustering iteration.

$$v = \frac{\sum_{j=1}^{|s|} d(t_j)}{|s|} \tag{3.7}$$

Step 5: Repeat step 3 and step 4 until the cluster centers no longer move.

Table 3.1: An example of failed tests execution with 7 statements and 8 failed test cases

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
m_1	1	1	1	1	1	1	1	1
m_2	1	0	0	1	1	0	1	1
m_3	1	0	0	1	1	0	0	0
m_4	0	1	1	0	0	1	0	0
m_5	0	1	1	0	0	1	0	0
m_6	0	0	0	0	0	0	0	0
m_7	1	1	1	1	1	0	1	1

To illustrate how failed test cases are clustered based on their execution profile similarity, refer to the example shown in Table 3.1. The table consists of eight failed test cases and seven statements. From step 1 and step 2, it is concluded that there are two clusters (i.e. $k = 2$) with test case t_2 assinged as cluster center v_1 (cluster center for the first cluster in iteration 1) and test case t_5 assinged as cluster center v_2 (cluster center for the second cluster in iteration 1) where the cluster centers are randomly assigned.

After estimating the number of clusters and assigning the cluster centers, step 1 and step 2 are completed. Moving to step 3, the Euclidean distance between a test case and the cluster centers will be calculated using Equation 3.4 and Equation 3.5. Therefore, the Euclidean distance between a test case and the corresponding cluster centers v_1 and v_2 for the first clustering iteration (iteration 1) is shown in Table 3.2.

Table 3.2: First k-means clustering iteration (Iteration 1)

$d(t, v)$	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
v_1	2	0	0	2	2	1	1.73	1.73
v_2	0	2	2	0	0	2.23	1	1
$\in s$	k_2	k_1	k_1	k_2	k_2	k_1	k_2	k_2

Based on Equation 3.6, if the minimum distance value is from v_2, then a test case t_j will belong to the second cluster k_2, and if the minimum distance value is from v_1, then a test case t_j will belong to the first cluster k_1. Therefore, k_2 has t_1, t_4, t_5, t_7, and t_8 failed test cases in this iteration, while k_1 has t_2, t_3, and t_6 failed test cases.

In other words, because the distance between t_1 and v_1 is higher than that between t_1 and v_2 ($2 > 0$), the same applied to t_4, t_5, t_7, and t_8, hence, t_1, t_4, t_5, t_7, and t_8 will be grouped in the second cluster (k_2). Correspondingly, the distance between t_2 and v_2 is higher than that between t_2 and v_1, the same applied to t_3 and t_6, hence, t_2, t_3, and t_6 will be grouped in the first cluster (k_1). In step 4, the mean point of each cluster center is recalculated and updated for cluster one and cluster two (k_1 and k_2). To update the mean point of the cluster centers v_1 and v_2, Equation 3.7 is computed based on the clustering results obtained in the first clustering iteration (Table 3.2). For v_2, the cluster center contains five test cases which are t_1, t_4, t_5, t_7, and t_8 which formulate k_2 for the first clustering iteration with each

test case having a distance value of 0, 0, 0, 1, and 1, respectively. Therefore, using Equation 3.7, the mean point of the new cluster center for the next clustering iteration for v_2 will be calculated as $v_2 = \frac{0+0+0+1+1}{5} = 0.4$. For v_1, the cluster center contains three test cases which are t_2, t_3, and t_6 which formulate k_1 for the first clustering iteration with each test case having a distance value of 0, 0, and 1, respectively. Therefore, the mean point of the new cluster center for the next clustering iteration for v_1 will be calculated as $v_1 = \frac{0+0+1}{3} = 0.33$. Henceforth, with the result of the new cluster centers as ($v_1 = 0.33$ and $v_2 = 0.4$), the clustering will be halted at the first iteration as the new mean point of v_1 and v_2 is not greater than one, so the new cluster centers cannot be set in the memory space of the program in Table 3.1. Therefore, the entire clustering process is complete.

3.2.3. Similarity Coefficients Metrics

Generally, the suspiciousness of a statement in P is directly proportional to the number of failed test cases that covered it. Therefore, less suspicious statements are statements that are mainly covered by passed test cases rather than failed test cases. However, for both passed and failed tests execution, the statements that are covered should make more contributions to fault localization than program statements that are not covered. Therefore, the statements that are covered by test cases are more influential and should carry more weight than the statements that are not covered by any test cases in suspiciousness computation. Many similarity coefficient-based fault localization techniques for suspiciousness computation have been proposed and rigorously evaluated by various empirical studies in the last decades (Abreu et al., 2007; J. A. Jones & Harrold, 2005).

In this study, three well-known similarity coefficient-based metrics, namely Ochiai coefficient, Naish2 coefficient, and Jaccard coefficient are chosen as shown in Table 3.3.

These three coefficients are chosen because they have shown to be very effective in locating software faults in recent years (Abreu et al., 2007, 2011; W. E. Wong et al., 2014). However, similarity coefficient-based metrics are not equivalent in performance for fault localization. As identified in an earlier study, their performance varies based on the scenarios they are used on. There is no one coefficient that will claim to be more effective in all scenarios (Yoo et al., 2014). However, these coefficients particularly Ochiai coefficient, have shown in the previous studies to be the most effective similarity coefficient metric (Abreu et al., 2007).

Table 3.3: Similarity coefficient metrics

Coefficient	Formula
Ochiai	$\frac{Ncf}{\sqrt{(Ncf + Nnf) \times (Ncf + Ncs)}}$
Naish2	$Ncf - \frac{Ncs}{Ncs + Nus + 1}$
Jaccard	$\frac{Ncf}{Ncf + Nuf + Ncs}$

Based on Table 3.3, *Ncf* denotes the number of failed tests that cover a statement, *Ncs* denotes the number of passed tests that cover a statement, *Nus* denotes the number of passed tests that do not cover a statement, and lastly, *Nnf* denotes the number of failed tests that do not cover a statement. The next section presents the proposed multiple-fault localization technique based on complex network theory (FLCN-M).

3.3. Multiple Fault Localization based on Complex Network Theory (FLCN-M)

In the study of complex network, various network centrality measures are introduced to determine the importance of a node in a network. For instance, measuring how important an individual is in a social network, identifying key and strategic nodes in the internet or urban networks (Dorogovtsev & Mendes, 2003), identifying important nodes and their correlation with faults in software programs (Zhu et al., 2011). Some of these centrality measures include degree centrality, closeness centrality, betweenness centrality, eigenvector and so forth.

For FLCN-M fault localization technique, two centrality measures are adopted for fault diagnosis, namely degree centrality and closeness centrality. Degree centrality measures the number of connections a node has to other nodes in a network, and closeness centrality of a node measures how close a node is to other nodes in a network. A new ranking formula is proposed to compute the suspicious values of program statements (Zakari, Lee, & Chong, 2018). The proposed technique ranks program statements based on their behavioral abnormalities and distance between statements in both passed and failed tests execution. This is based on the knowledge that a faulty statement might play a distinct role in the network. Furthermore, the technique will aid the developer in localizing multiple faults simultaneously in a single diagnosis rank list.

In this thesis, statements and nodes are used interchangeably while execution between program statements is represented as edges.

Definition 1. Degree centrality. Degree is a commonly used centrality measure in complex network and it can be utilized to statistically measure node importance in a given network (C. Gao et al., 2013). Freeman and Linton stated that the degree of a given node is measured by the number of adjacencies the node has in a network (L. C. Freeman,

1978). In other words, is the total connection a given node has to its neighboring nodes in a network. Degree centrality, *Dc*(*i*), for node *i* can be formalized as stated in Equation 3.8 (L. C. Freeman, 1978).

$$Dc(i) = \sum_{j}^{n} x_{ij} \qquad (3.8)$$

where *i* is the focal node, *j* represents any other neighbor node, *n* represents the total number of nodes in the network, and *x* is the adjacency matrix, in which x_{ij} is indicated as 1 if there is a connection between node *i* and node *j*, and 0 otherwise as stated in Equation 3.9. The connection between nodes is also known as the edge between nodes.

$$x_{ij} = \begin{cases} 1, & connection, \\ 0, & Otherwise \end{cases} \qquad (3.9)$$

Given a sample network in Figure 3.5, with four nodes and four edges. In this network, based on the adjacency matrix, node 1 will have one edge to node 2 and node 2 will have three edges to node 1, node 3, and node 4. Node 3 will have two edges to node 2 and node 4, while node 4 will also have two edges to node 3 and node 2. Therefore, the adjacency matrix measure the level of connection each node has in the network at the local level. In this example, node 2 is regarded as the most influential in the network.

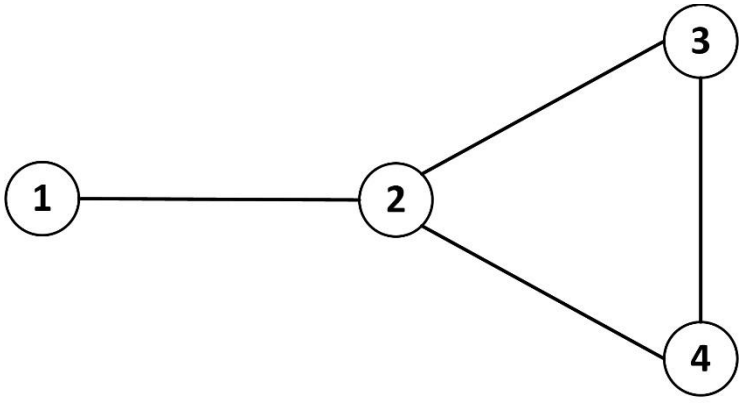


Figure 3.5: An example of a network consisting of 4 nodes and 4 edges

In this context, a network *N* will be modeled based on software tests execution profile in correspondence to the statements' executed (Section 3.3.1). Therefore, for a statement m_i in a sparse undirected/unweighted network, if there is a connection between the statement m_i to statement m_j in a test case execution, meaning that there is an edge between the two statements. Hence, if statement m_i has an edge from another statement m_f that is executed by a separate test case, then, statement m_i will have two edges because it is connected to two executable statements (m_j and m_f) executed by different test cases. The study in (Opsahl & Panzarasa, 2009) deduces that a node with a higher degree centrality is likely to be stronger connected in a given network, hence, being more likely to be the cause of failure or related to failure. This centrality measure is aimed to identify behavioral abnormalities in statements executions and identify the most central statements in the program network. Therefore, in this thesis, Equation 3.8 can be represented as Equation 3.10 (L. C. Freeman, 1978).

$$Dc(m_i) = \sum_{j=1}^{n} x(m_i, m_j) \qquad (3.10)$$

where m_i is the focal statement, m_j represents any other neighbor statement, *n* represents the total number of statements, and $x(m_i, m_j)$ is the adjacency matrix. The calculation of statements' degree centrality (*Dc*) in a program network *N* is computed by Equation 3.10.

Therefore, this centrality measure will aid in identifying program statements that are the most central and the most important in *N*. However, as identified by Chen et al., higher degree centrality does not always point to how influential a node is (Chen et al., 2012). Therefore, to help in diagnosing statements that are more suspicious in *N*, closeness centrality measure is also adopted.

Definition 2. Closeness centrality. Closeness centrality measures the inverse of the average shortest path between a node and all other nodes in *N*, which means that all paths should lead to a node (Cheng & Suthers, 2011; Šubelj & Bajec, 2012). This centrality measure tries to measure how long it will take to spread information, diseases, or failures from the node of interest to all other nodes sequentially. For fault that propagates through multiple program statements or faulty statements that are close to each other in the program, this centrality measure will aid in identifying them. Therefore, statements that are closer to the statements with abnormal behaviors (relatively high degree centrality) in *N* will be identified because the higher the closeness centrality value of a statement, the closer it is to all other statements. As shown in Figure 3.6, program statements m_3 and m_5 might have distinct degree centrality, but due to their close proximity, they will have a relatively close closeness centrality value with respect to the tests execution.

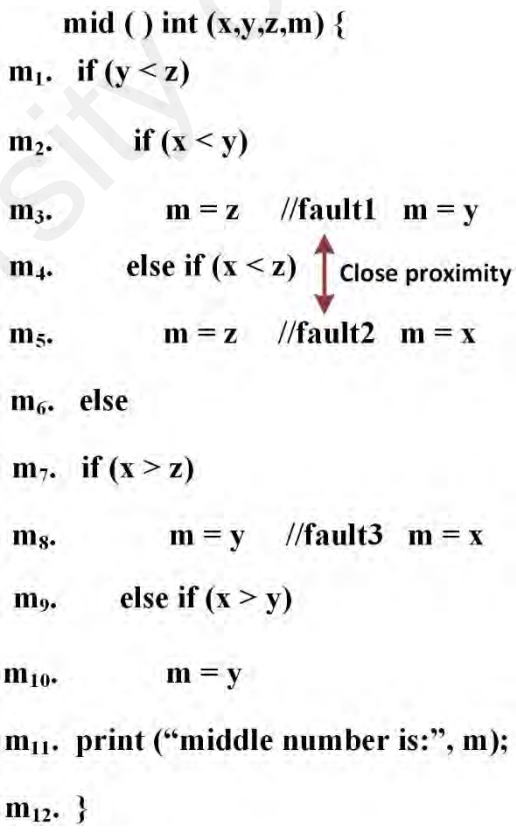


Figure 3.6: Closeness centrality example for program mid ( )

Therefore, the closeness centrality of a statement m_i to all other statements is computed using Equation 3.11(Cheng & Suthers, 2011).

$$Cc(m_i) = \frac{n-1}{\sum_{j \neq i}^{n} d(m_i, m_j)} \qquad (3.11)$$

where $d(m_i,m_j)$ is the shortest path distance between statement m_i and m_j, and n is the total number of statements in N.

Definition 3. Suspicious score. To further calculate the suspiciousness S of statement m_i in N, a new ranking formula is proposed as shown in Equation 3.12. The technique FLCN-M computes the suspicious value of m_i using the degree centrality value Dc of m_i and its closeness centrality Cc value in N. For a given program statement m_i, the difference between the two values will be computed using Equation 3.12. Computing the difference of these values will give a developer quantifiable value of how suspicious a statement is. The first item at the left side of Equation 3.12, i.e. Dc, indicates how central a program statement is in N. In other words, it will aid in knowing how connected a program statement is to other program statements in both passed and failed tests execution in N. As for the second item on the right, i.e. Cc, it helps in identifying the program statements that are closer to other statements sharing common behavior in N. In other words, it aids in knowing how close a program statement is to other statements in N. Therefore, knowing the difference between the two values for program statements will aid in identifying faulty program statements based on this behavioral abnormalities in both passed and failed tests execution. Furthermore, the difference of these values will also help in ranking multiple faulty statements that are closer to each other or caused by the same failure or faults revealing variables (Zakari, Lee, & Chong, 2018).

$$S(m_i) = Dc_i - Cc_i \qquad (3.12)$$

For *n* number of program statements modeled as *N*, the suspicious score value will be assigned for each statement in *N*. The program statements will be generated in descending order of their suspicious scores. Henceforth, a developer will start checking the program statement with the highest suspicious score until the faulty program statements are identified. Appendix B shows some of the results generated by FLCN-M.

3.3.1. Network Modeling

To model a network *N*, a sample program given in Table 3.4 is used. In the program, if a statement execution is labeled as 1, it signifies that the statement is executed by the test case in that test run, and 0 otherwise. For the test result of each test case, 0 means the test case has passed while 1 means the test case has failed. The program has 12 statements (*n* =12) with 11 executable statements and six test cases. Table 3.5 illustrates the network construction sequence, a single network *N* is modeled to capture the entire program statements execution behavior.

Table 3.4: A multiple-fault program with tests execution

	mid ( ) { input x, y, z, m;	t_1	t_2	t_3	t_4	t_5	t_6
m_1	if (y < z)	1	1	1	1	1	1
m_2	if (x < y)	1	1	1	1	1	1
m_3	m = z; //fault 1 m = y	1	1	0	0	1	1
m_4	else if (x < z)	0	1	0	0	0	0
m_5	m = z; //fault 2 m = x	1	0	0	0	1	1
m_6	else	1	0	0	0	0	1
m_7	if (x > z)	0	0	1	1	0	0
m_8	m = y; //fault 3 m = x	1	0	1	1	0	1
m_9	else if (x > y)	0	0	1	0	0	0
m_{10}	m = y	0	0	0	1	0	0

Table 3.4, continued

m_{11}	print ("middle number is:", m);	0	0	0	0	0	0
m_{12}	}	0	0	0	1	1	0
	Pass/Fail Status	0	0	0	0	1	1

Cytoscape software platform (http://www.cytoscape.org/) is used for network construction and generation. The network is modeled as an undirected and unweighted network. To build a software-based complex network, a developer needs to have a valid input regarding the structural behavior of the software such as software components and their relationships. In this thesis, the input values will be the program spectra and the execution results (passed/failed) of a program.

Table 3.5: Network construction sequence

Test cases	Execution trace	Test result
t_1	{1-2}{2-3}{3-5}{5-6}{6-8}	Passed
t_2	{3-4}	Passed
t_3	{2-7}{7-8}{8-9}	Passed
t_4	{8-10}{10-12}	Passed
t_5	{5-12}	Failed
t_6		Failed

Execution profile of {t_1, t_2, t_3, t_4, t_5, and t_6} is used to model the network N irrespective of their execution results (passed/failed). From t_1, there is an edge from m_1 to m_2, m_2 to m_3, m_3 to m_5, m_5 to m_6, and m_6 to m_8, respectively. From the second test case t_2, an edge is added from m_3 to m_4. For t_3, an edge is added from m_2 to m_7, m_7 to m_8, and m_8 to m_9. For t_4, there is an edge from m_8 to m_{10}, and m_{10} to m_{12}, while for t_5, there is an edge from m_5 to m_{12}. Therefore, in a case where there are two test cases with similar execution profile, one test case will be chosen and the other test case will be discarded because the test cases have the same execution path. That is why test case t_6 was not included in the network construction sequence in Table 3.5 because it has the same execution profile as test case t_1. Statement

m_{11} will not be modeled because the statement was not executed by any test case. Therefore, all the test case executions corresponding to the statements will be modeled as *N*.

3.3.2. General Framework

This section presents the detailed steps of the proposed technique, FLCN-M, in locating multiple faults. Figure 3.7 shows the overall fault localization process.

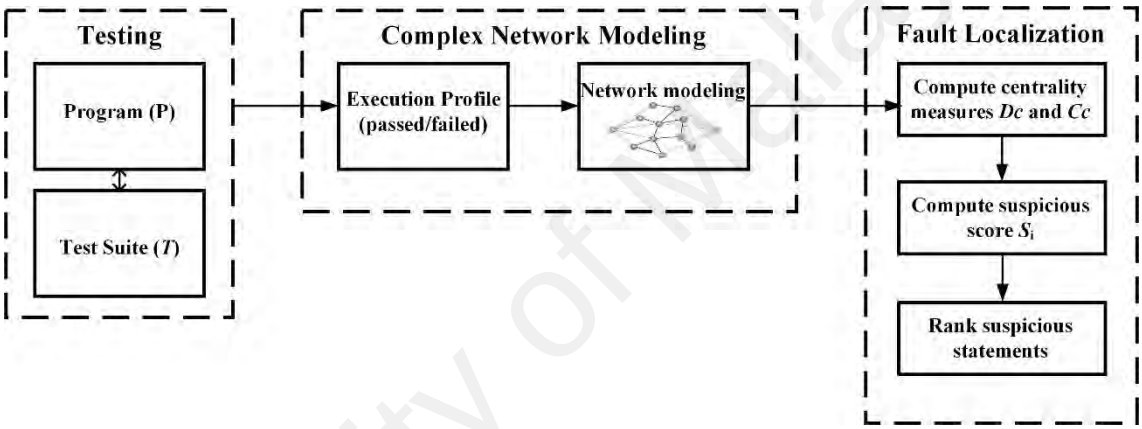


Figure 3.7: FLCN-M fault localization process

- **Step 1:** In this step, the faulty program P will be executed by all the available test cases in *T* and the execution data will be collected. The execution profile of statements with respect to each test case will be collected. The set of passed and failed test cases will be identified.

- **Step 2:** A network *N* will be modeled using the execution profile of both passed and failed test cases as input based on the process detailed in Section 3.3.1.

- **Step 3:** For *n* number of program statements modeled as *N*, the *Dc* and *Cc* of each statement will be computed based on Equation 3.10 and Equation 3.11.

Furthermore, Equation 3.12 will be computed to calculate the suspicious score value S of each statement in N.

- **Step 4:** Based on the suspicious score values of program statements, rank the statements $m_1, m_2, m_3, \ldots, m_n$ based on $S_1, S_2, S_3, \ldots, S_n$ in descending order of their suspicious score values. A developer will be tasked to examine the statements one by one from the top until all faults are located. Even if the developer locates the first fault, the localization process will continue until all the faults are located in the single diagnosis ranking list.

3.3.3. A Running Example

Consider the sample program in Table 3.4, which takes three integers as input to demonstrate how FLCN-M can be used to locate multiple faults. The program has 12 statements (n =12) with 11 executable statements and 3 faults in statements m_3, m_5, and m_8. The faulty statement m_3 is executed by two passed test cases, t_1 and t_2, and two failed test case, t_5 and t_6. Faulty statement m_5 is executed by one passed test case t_1 and two failed tests, t_5 and t_6, and m_8 is executed by three passed tests, t_1, t_3, and t_4, and one failed test case t_6. The proposed technique will rank the faulty statements at the top of the ranking list because it takes into consideration behavioral abnormalities and distance between statements irrespective of whether they are executed by passed or failed tests execution.

At step 2, to model the network N, the process detailed in Section 3.3.1 will be used. Thus, the network as shown in Figure 3.8 is generated. Step 1 and step 2 are completed.

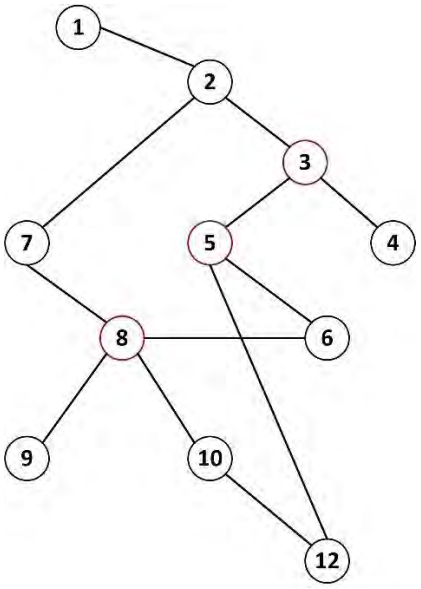


Figure 3.8: Network for the multiple-fault program mid ( )

Moving to step 3, the degree centrality *Dc* of a statement m_i and the closeness centrality *Cc* of a statement m_i to all other statements in the network *N* are calculated. The suspicious score *S* of the program statements will be calculated according to Equation 3.12. Next, in step 4, rank the statements $m_1, m_2, m_3, \ldots, m_{12}$ based on $S_1, S_2, S_3, \ldots, S_{12}$ in descending order of their suspicious score values. The developer will examine the statements one by one from the top until the multiple faults are located. The fault examination process will not be halted even if the developer locates the first fault, the localization process will continue until all the faults are located in the single diagnosis ranking list. The localization result is shown in Table 3.6.

Table 3.6: Localization result of program mid ( ) with multiple faults

m_i	Dc_i	Cc_i	S_i	Rank
m_1	1	0.333	0.667	7
m_2	3	0.476	2.524	2
m_3	3	0.476	2.524	2
m_4	1	0.333	0.667	7
m_5	3	0.5	2.5	3

Table 3.6, continued

m_6	2	0.454	1.546	5
m_7	2	0.476	1.524	6
m_8	4	0.5	3.5	1
m_9	1	0.345	0.655	8
m_{10}	2	0.417	1.583	4
m_{12}	2	0.417	1.583	4

In this example, FLCN-M ranks the statements based on their suspicious score. After ranking, the following ranking is obtained m_8, m_3, m_2, m_5, m_{10}, m_{12}, m_6, m_7, m_4, m_1, and m_9. This shows that m_8 is more likely to contain faults even though the statement is executed by three passed test cases and only one failed test case. Therefore, the first fault can be located by examining the first statement in the ranking list. The second and third faults can be found when m_3 and m_5 are examined. However, looking at Table 3.6, all the faulty statements have relatively higher *Dc* and *Cc* values. In total, based on this example, FLCN-M technique can locate all the three faults by examining less than 4 statements in a single diagnosis rank list.

3.4. Single Fault Localization based on Complex Network Theory (FLCN-S)

To improve effectiveness in a single fault context, a new fault localization technique that leverages failed tests execution alone in the context of a single fault, named FLCN-S, is proposed. This technique adopts all the centrality measures, ranking formula, and process used in the former technique (FLCN-M) detailed in Section 3.3. However, the network modeling process is different where instead of using both passed and failed tests execution, failed tests execution alone are utilized for the network modeling process. Equation 3.10 and Equation 3.11 will be used for fault diagnosis, while Equation 3.12 will be used to compute the suspicious scores of all the program statements. Because for a program with a single fault, the faulty statement is more likely to be executed by failed

test cases due to less fault-to-failure complexity as shown in previous studies (Abreu et al., 2007; DiGiuseppe & Jones, 2015; J. A. Jones & Harrold, 2005).

To demonstrate the technique (FLCN-S), the single-fault version of the program in Table 3.4 is taken with only the failed tests execution profile as shown in Table 3.7. Consider that the program has a single fault in m_5 with two failed test cases t_5 and t_6. Using FLCN-S, the network *N* will be modeled by utilizing the failed tests execution only.

Table 3.7: Program mid ( ) with a single fault

mid ( )	t_5	t_6
m_1	1	1
m_2	1	1
m_3	1	1
m_4	0	0
m_5	1	1
m_6	0	1
m_7	0	0
m_8	0	1
m_9	0	0
m_{10}	0	0
m_{11}	0	0
m_{12}	1	0
Fail Tests	1	1

Therefore the network in Figure 3.9 is generated using the above-failed tests execution. For the first test case t_5, there is an edge from m_1 to m_2, m_2 to m_3, m_3 to m_5, and m_5 to m_{12}. For the second failed test t_6, there is an extra edge from m_5 to m_6 and m_6 to m_8 respectively. The faulty statement m_5 is executed by all the failed test cases.

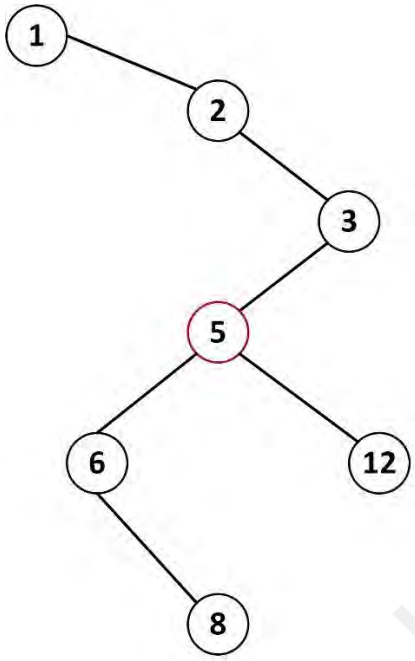


Figure 3.9: Network for a single-fault program mid ( )

The *Dc* and *Cc* of all the statements will be calculated. Next, the suspicious score *S* of each statement will be calculated to generate the suspicious statements ranking list. Lastly, the localization result is shown in Table 3.8. The developer will be tasked to check the program statements in descending order of their suspicious scores until the faulty statement is located.

Table 3.8: Localization result of program mid ( ) with single fault

m_i	m_1	m_2	m_3	m_5	m_6	m_8	m_{12}
Dc_i	1	2	2	3	2	1	1
Cc_i	0.316	0.428	0.545	0.6	0.461	0.333	0.4
S_i	0.684	1.572	1.455	2.4	1.539	0.667	0.6
Rank	5	2	4	1	3	6	7

Looking at the result in Table 3.8, the faulty statement m_5 is ranked at the top of the ranking list with the highest suspicious score, which means that the statement is most

likely to contain the fault. Therefore, the fault localization process will be halted. Appendix A shows some of the results generated by FLCN-S.

3.5. Community-based Fault Isolation Approach for Simultaneous Fault Localization

In this section, the network community clustering algorithm used for fault isolation is outlined. Furthermore, the proposed community weighting and selection process which will help in selecting and prioritizing *fault-focused* communities for effective simultaneous localization of faults in parallel is also highlighted. Finally, the general framework for the proposed approach is also highlighted with a running example to illustrate the approach.

3.5.1. Community Clustering Algorithm

To cluster failures in the fault localization domain, various clustering algorithms were utilized in recent years (R. Gao & Wong, 2017; Högerle et al., 2014; James A Jones et al., 2007; Yabin Wang et al., 2014). These studies mainly use program tests execution profile to isolate faults into distinct clusters with tests execution profile similarity often used to justify failure groupings. Hence, distance metrics such as Euclidean distance, Jaccard distance, and Hamming distance are often used to compute the test-to-test distance to determine the cluster into which a given test will fall (Huang et al., 2013; James A Jones et al., 2007; W. E. Wong, Debroy, Golden, et al., 2012). However, recent studies have shown that the use of this representation is problematic and is not an effective way to isolate failed tests based on their causative faults (R. Gao & Wong, 2017; C. Liu

et al., 2008). In contrast, for the proposed approach, the network *N* is the representation of all the program tests execution profile of both passed and failed tests execution. Therefore, this approach does not perform clustering on tests execution rather it performs clustering on program statements in the network *N* that is modeled based on the tests execution. Therefore, the *due-to* relationship between program statements is measured to create *fault-focused* communities instead of between failed test cases as done in the existing works. Instead of using the traditional distance metrics which are somewhat less effective for measuring the *due-to* relationship between tests, edge-betweenness based distance is used by the divisive network community clustering algorithm to measure the distance between program statements executions. Network community clustering algorithms fall into two classes which are agglomerative and divisive (Scott, 2000). Algorithms are classified based on whether they concentrate on addition or removal of edges to a network or out of a network.

In this study, the latter (divisive) clustering method is utilized to isolate faults into different communities. Based on this method, a developer is tasked to find the least connected statements (statements with the highest edge-betweenness score) in a faulty program network and then remove the edges between them. If this process is done repeatedly, the program network will naturally be divided into smaller and smaller groups composed of densely connected statements. These smaller groups can be considered as the network communities when the process is halted. This is based on the knowledge that the more densely connected nodes are in a network, the more prone they are to be related to the same variable or information (Blondel, Guillaume, Lambiotte, & Lefebvre, 2008; Šubelj & Bajec, 2011). In extension to this study, it is postulated that the more densely connected program statements are in the program network *N*, the more prone they are to be related to the same failure or fault. The approach to community identification in this

study basically follows these lines. The algorithm proposed by Girvan and Newman for community detection is used (Girvan & Newman, 2002; Newman & Girvan, 2004).

Communities are generated by continuously removing edges (connection between program statements) from the modeled program network based on their betweenness centrality score value. Node betweenness centrality is defined as a measure of nodes centrality and influence in a network (L. C. Freeman, 1977). The betweenness centrality measures node influence based on information flow from a node to other nodes in a network, particularly where the information flow in the network follows the shortest available paths. Hence, to identify the statements in a program network *N* that are mostly between other statements, the betweenness centrality is generalized to network edges as edge-betweenness. Edge-betweenness of an edge is defined as the number of the shortest paths between statements that run along it. For statements that have more than one shortest path, all the paths will be given equal weight so that the total weight of all paths is unified. In a scenario where a program network has communities or clusters that are loosely connected by intergroup edges, the shortest paths between these communities must move along these few edges. Therefore, edges connecting communities will normally have high edge-betweenness score values. The community structure of the program network can be revealed in distinct groups by removing these edges.

The algorithm used in this study for community detection in its general form is stated as follows:

1) Calculate the betweenness scores of all edges in the network.
2) Identify and remove the edge that has the highest betweenness score.
3) Recalculate betweenness scores for all the remaining edges affected by the removal.
4) Repeat from step 2 until no edges remain.

To calculate the betweenness of a program network, Newman fast algorithm is used (Newman, 2001). This algorithm calculates the betweenness for all the edges e in a network of n nodes in best-case time O(en) as claimed in (Newman & Girvan, 2004). The betweenness of edges that are affected by the removal of the edge in step 2 will be recalculated. Therefore, this algorithm calculates the betweenness score of each edge starting from statement in N until the betweenness score is computed from each and every statement in N. Henceforth, the betweenness scores of the edges from each and every statement will be added up and divided by 2 to get the final edge-betweenness scores of all the edges in N. To get a better result, the recalculation step of the algorithm is the most vital step. Therefore, the recalculation step is very crucial in detecting good communities in the program network N.

3.5.2. Shortest-path Betweenness

In this study, to calculate the shortest path between program statements in a program modeled network N, it can be done using breadth-first search in time O(e) with O(n^2) statements pairs (Leiserson, Rivest, Stein, & Cormen, 2001; Newman, 2001). Breadth-first search can find the shortest paths from a statement m_i to all other statements in time O(e). Figure 3.10 shows an example of a shortest path "tree" for a simple network. Figure 3.10a shows a simple network that illustrates how breadth-first search finds the shortest paths between statements in time O(e) where e represents the edge between statements. The number of shortest paths from statement m_a to itself is weight $w_a = 1$ as an initial condition. For any other statements that are directly next to m_a, in this case, m_b and m_e, they will be given equal weight as m_a. In the case of m_f, the number of shortest paths from m_a to m_f is the number of shortest path from m_a to m_b plus the number of shortest path from m_a to m_e. Therefore, m_f will carry the weight of 2 as $w_f = 2$. In other words, because

m_a has multiple paths to m_f and each path holds a weight of 1. Therefore, the weight of m_f will be the weight of m_a to m_b plus the weight of m_a to m_e which equals to a total weight of $w_f = 2$ for m_f.

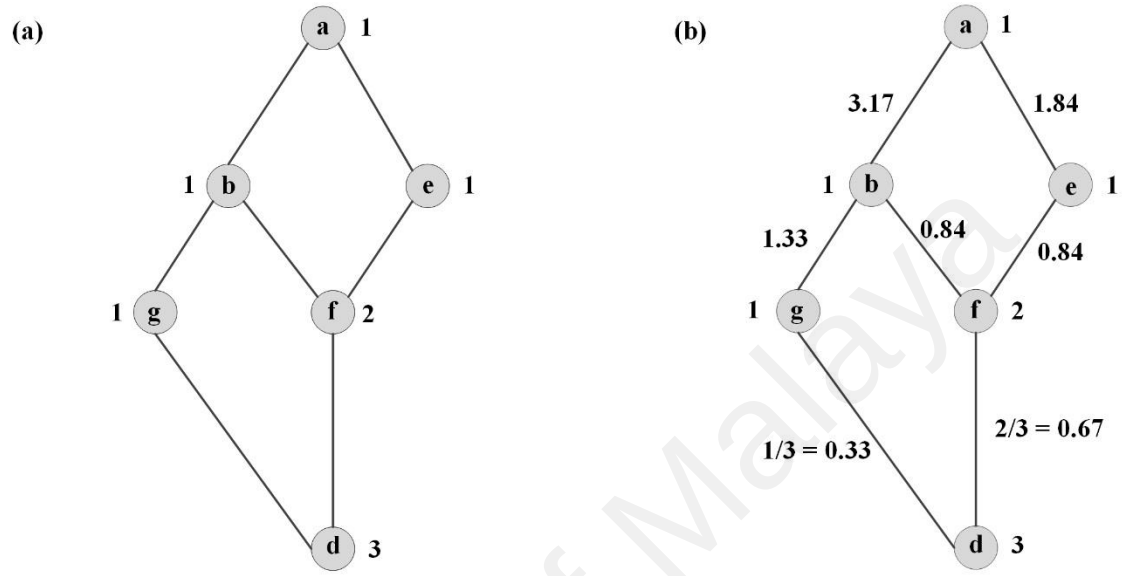


Figure 3.10: Calculation of shortest-path betweenness

However, the number of shortest paths from m_a to m_b is the same as the number of shortest paths from m_a to m_g, because one has to go through the predecessor m_b in a single direction. Next, to know the number of shortest paths from m_a to m_d (the lowest statement) one have to sum up the number of shortest paths from m_a to m_g and the number of shortest paths from m_a and m_f. So, m_d will carry the weight of 3 ($w_d = 3$). Now the shortest path of the statements can be used to calculate the betweenness for each edge in the network. Figure 3.10b shows the betweenness score of each edge in the network. First, one has to start with the edges that are farthest from the initial statement (m_a) which are the lowest edges. Then, work upwards assigning a score to each edge that is 1 plus the sum of the scores on the edge or edges immediately below it. For example, starting from the edge of the farthest statement (m_d) in Figure 3.10b, the betweenness scores of the edges from m_d to m_g and from m_d to m_f will be calculated by dividing the weight of m_g and m_d and the same applied to m_f and m_d. When one has go through all edges in the network, the

resulting scores of the edges are the betweenness scores for the paths from m_a (the betweenness scores of the edges are to be calculated from all the remaining statements as well). This process will be repeated for all statements, and the betweenness scores of all edge will be added up and divided by 2. With this, the final edge-betweenness scores for the shortest paths between all statements will be obtained.

The breadth-first search and the process of calculating the betweenness of all edges in the network both takes worst-case time O(e). With n statements in total, therefore the whole calculation is done in best-case time O(en) as claimed (Newman & Girvan, 2004). In an earlier definition of node betweenness (L. C. Freeman, 1977), if a node has multiple shortest paths (i.e. m_f in Figure 3.10), all paths leading to the node will be given equal weights summing to 1. For instance, if there are two shortest paths, each path will be given the weight of $\frac{1}{2}$.

Looking at Figure 3.10a, to conduct a breadth-first search starting from m_a, the following steps will be performed:

1) The initial Statement m_a will be assigned a distance $d_a = 0$ and a weight $w_a = 1$.
2) For any statement m_i that is next to m_a, a distance $d_i = d_a + 1 = 1$, and a weight $w_i = w_a = 1$ will be given to it.
3) Next, for each statement m_j next to the statement m_i, the following three things are conducted:
    - If m_j is not assigned any distance, a distance $d_j = d_i + 1$ will be assigned and a weight $w_j = w_i$ will be assigned.
    - Moreover, if a distance has already been assigned to m_j as $d_j = d_i + 1$, then, the weight of the statement will be rise by w_i which is $w_j \leftarrow w_j + w_i$.

- If m_j has already been assigned a distance and $d_j < d_i + 1$, nothing will be done.

4) Lastly, repeat from step 3 until no statement left that has been assigned a distance but its neighboring statement distance is not assigned.

Furthermore, the weight of a statement m_i indicates the number of different paths from the root statement m_a to m_i. Therefore, these weights are exactly what is needed to calculate the edge-betweenness where for two statements m_i and m_j that are connected, with m_j farther than m_i from the root statement m_a, then the shortest paths from m_j through m_i to the root statement m_a will be given by w_i/w_j. To further calculate the edge-betweenness from all the shortest paths starting from m_a as shown in Figure 3.10b, the following steps will be carried out:

1) Calculate the shortest paths starting from statement m_a, to every other statement using the breath-first search in time O(e).
2) Find the lowest statement m_j (i.e. statement at the bottom of the program network N)
3) For each given statement m_i next to statement m_j, assign a score to the edge from m_i to m_j of w_i/w_j.
4) Next, start from the edges that are far away from the root statement m_a (i.e. statements edges that are at the bottom of the diagram in Figure 3.10) moving upward. For the edge of m_i to m_j, with m_j being far away from m_a than m_i, a score that adds 1 to the sum of the neighboring edges immediately below it will be assigned (i.e. edges below it that share common statements). At last, the scores will be added up by the weight w_i/w_j.
5) Lastly, repeat step 3 and step 4 until m_a is reached.

Using these steps, one is able to calculate the edge-betweenness of all edges in the program network in best-case time O(*en*). This calculation will be repeated for each edge removed from the program network *N*. Given that there is *e* amount of edges where $e = \{e_1, e_2, e_3, \ldots, e_n\}$. The whole community structure algorithm based on the shortest path betweenness will run in worst-case time $O(e^2n)$ or $O(n)^3$. It is observed that unlike networks with stronger community structures in other research domains (Dorogovtsev & Mendes, 2003; Newman & Girvan, 2004), a network modeled based on tests execution profile (program spectra) has less community density (mostly sparse). Therefore, the time O(*en*) it takes to generate *fault-focused* communities is relatively long but has less effect on the quality of communities created for fault localization.

3.5.3. Community Weighting and Selection

In practice, when a program fails, a developer does not normally know the number of faults that caused the failure. In extension, when identifying network communities (clusters), the developer probably does not know how many communities the algorithm is going to generate depending solely on network to network modularity (Newman & Girvan, 2004). Therefore, there is no reason for the identified communities to be roughly the same size. For the above practical scenario where the developer does not know the exact number of faults and the exact number of communities or their sizes, localizing those faults might be a bit tricky. Therefore, a community weighting and selection mechanism is introduced to aid in prioritizing highly important communities to the available developers to debug the faults simultaneously in parallel. In this work, these communities are named as *fault-focused* communities which target a single fault each.

For a given community *C* in a network *N*, a weight will be assigned to each community based on the total number of statements *n* a given community contains. Equation 3.13 will calculate the weight of a single community in network *N*.

$$C = \sum_{j=1}^{n} m_j \qquad (3.13)$$

where *C* represents a community, *n* represents the total number of statements in that community, and *m* represents a statement. Therefore, the number of communities can be represented as $C = \{c_1, c_2, c_3, \ldots, c_n\}$ in a given program network *N*.

Furthermore, the weights of all the communities in *N* will be computed. Suppose there is a network *N* with three identified communities as shown in Figure 3.11 where the first community has 7 statements, the second community has 6 statements, and the third community has 4 statements. Therefore, if a given community has a higher weight with a greater number of statements in comparison with other communities in *N*, that community will be ranked at the top of the community ranking list followed by subsequent communities. All communities will be generated in descending order based on the weights they carry and will be given to developers to debug the faults simultaneously in parallel starting with the community with the highest weight. The main reason why starting with the community with the highest weight is because it is postulated that for communities with a high number of statements, they are more susceptible to contain the faulty program statements.

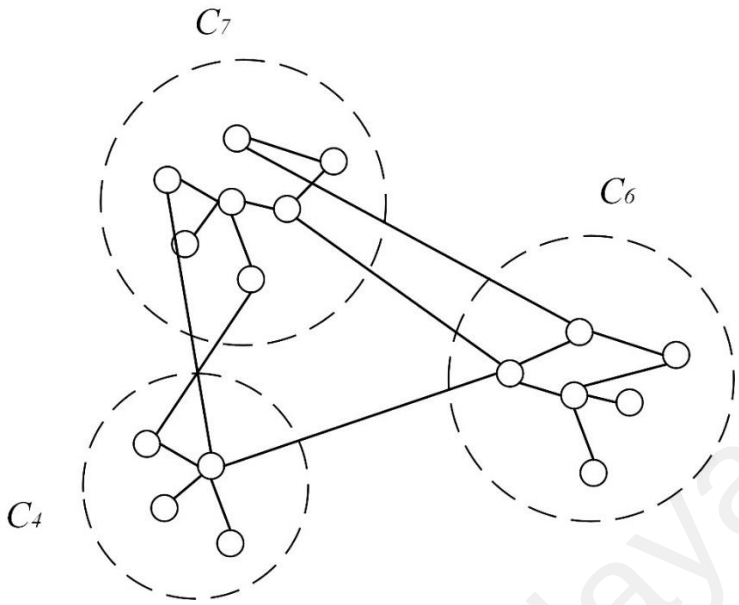


Figure 3.11: Network with groups of communities

3.5.4. The Community-based Fault Isolation Approach

In this section, the detailed steps of the proposed community-based fault isolation approach are presented. Figure 3.12 demonstrates the overall process of the approach, with more details of the process given as follows:

Step 1: *Program tests execution profile*: The first phase of the approach focuses on program execution and collecting the tests execution profile. The faulty program P under test will be executed with the corresponding available test cases to generate the tests execution profile (program spectra). Test cases executions are classified into passed and failed categories depending on whether the output deviates from the expected output or not. Therefore, if a test case produces an expected output, the test case has passed. Otherwise, the test case has failed.

Step 2: *Network modeling*: In this phase, the network *N* is modeled. The execution profile (program spectra) of both passed and failed test cases obtained from the initial

phase will be used as an input to generate the network *N* that captures the entire program execution behavior.

Step 3: *Community (cluster) detection*: The divisive network community clustering algorithm in Section 3.5.1 will be computed at this stage. Statements that are densely connected with each other will be isolated into distinct *fault-focused* communities by taking into account the statements edge-betweenness distance as discussed in Section 3.5.2. Therefore, the number of existing communities in a given network will be known by the developers.

Step 4: *Community (cluster) weighting and selection*: After knowing the number of communities in *N*, a developer needs to know where to start the debugging task which can be tricky especially if there are many available communities. Furthermore, not all the communities might contain faulty program statements. Therefore, the number of communities can possibly be larger than the number of faults or vice versa. In this stage, the community weighting and selection mechanism in Section 3.5.3 will be used to identify and rank the most fault-prone communities with high possibility of containing faulty statements. The *fault-focused* communities will be generated in descending order based on the weights they hold for developers to work with and localize the faults simultaneously in parallel.

Step 5: *Fault localization:* The fault localization technique based on complex network theory will be used for fault localization. Ultimately, faults will be found and neutralized by each developer. The program will be retested again to see if the debugging is successful.

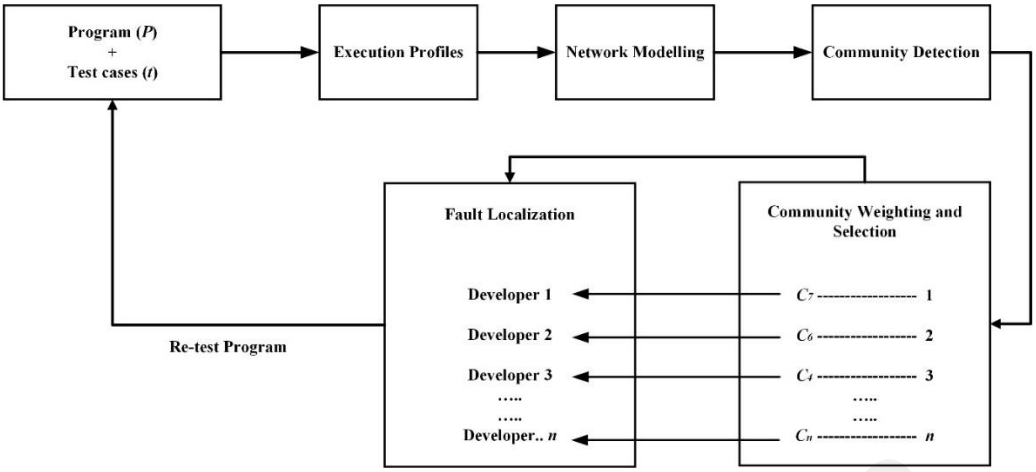


Figure 3.12: General framework of the proposed approach

3.5.5. A Running Example

Consider a running program sample in Table 3.9 for discussing how the approach will work for fault localization. The program has eight statements and is executed with five test cases. Statements m_2 and m_5 are both faulty with two failed test cases (t_2 and t_3) and three passed test cases (t_1, t_4 and t_5), respectively.

Table 3.9: A program with tests execution

	mid ( ) { input x, y, z, m;	t_1	t_2	t_3	t_4	t_5
m_1	if (y < z & x< y){	1	1	1	1	1
m_2	m = z; //fault 1 m = y	1	1	1	1	1
m_3	else	1	1	1	1	1
m_4	if (x > z)	1	0	1	1	1
m_5	m = y; //fault 2 m = x	0	0	1	1	1
m_6	}	1	0	0	0	1
m_7	print (“middle number is:”, m);	0	0	0	0	0
m_8	}	0	1	1	1	0

In the next step, the network N will be modeled so as to capture the entire program execution behavior on both passed and failed tests execution. For the first test case t_1, there is an edge from m_1 to m_2, m_2 to m_3, m_3 to m_4, and m_4 to m_6. For the second test case t_2, additional edge from m_3 to m_8 is added. For the third test case t_3, an edge is added from m_4 to m_5 and from m_5 to m_8. Test case t_4 will be ignored because it has the same execution profile with t_3. Therefore, all the resulting edges will be redundant. For t_5, an edge from m_5 to m_6 is added. Therefore, step 1 and step 2 are completed.

Moving to step 3, the divisive network community detection algorithm in Section 3.5.1 will be computed to identify *fault-focused* communities that target single fault each by taking into account the statements' edge-betweenness distances as discussed in Section 3.5.2. Moreover, Figure 3.13 shows the calculation of shortest path betweenness of all edges from all the program statements in the program network N of the program in Table 3.9. The figure also shows the process and results of calculating the breadth-first search and the process of calculating the betweenness scores of all edges from all the seven executable statements that the network contains. As stated earlier in Section 3.5.1 and 3.5.2, all these calculations will be repeated from each and every statement in N. For clarification, looking at Figure 3.13, the program network is divided into seven sub-networks, each network shows the betweenness score calculation for each edge from each statement in N. Therefore, the betweenness scores of all edges from all the statements in the respective networks will be added up and divided by 2 to get the final edge-betweenness score of an edge in N. From Figure 3.13, the betweenness scores of the edge from statement m_3 to m_4 in all calculations of each statement in N can be added up as 2.16 + 2.16 + 2.16 + 3.33 + 1.5 + 1.5 + 0.83 = 13.64. Therefore, the total score will be divided by 2 to get the final edge-betweenness score as (6.82) of that edge. The same is applied to all the remaining edges.

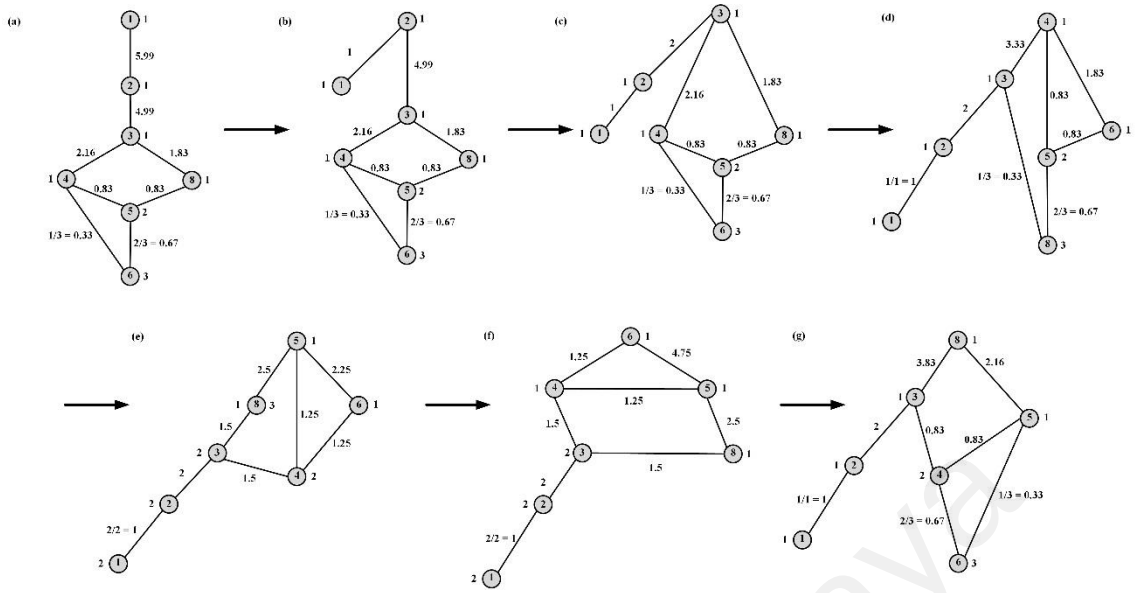


Figure 3.13: Calculation of shortest-path betweenness

Table 3.10 highlights the final edge-betweenness score for all edges in Figure 3.13 by adding up the betweenness scores of individual edges from all statements in N and dividing the total score by 2. It was identified that edge of m_2 to m_3 has the highest edge-betweenness score as 4.99 + 4.99 + 2 + 2 + 2 + 2 + 2 = 19.98. Hence, the final edge-betweenness score of the edge will be (19.98/2 = 9.99). Therefore, the edge will be removed to create two *fault-focused* communities, C_1 and C_2, where each community contains n number of statements. The *fault-focused* communities and their statements are, C_1 = {m_1 and m_2} and C_2 = {m_3, m_4, m_5, m_6, and m_8}.

Table 3.10: Final edge-betweenness score for each edge in the network

Edges	m_1 - m_2	m_2 - m_3	m_3 - m_4	m_4 - m_5	m_3 - m_8	m_4 - m_6	m_8 - m_5	m_5 - m_6
Edge-betweenness score	5.99	9.99	6.82	3.32	6.32	2.99	5.16	5.08

Next, based on the community weighting and selected process in step 4, the second community C_2 will be ranked at the top of the community ranking list because it has the

highest number of statements. Therefore, each of these communities will contain a single fault where community one (C_1) has faulty statement m_2 and community two (C_2) has faulty statement m_5. Lastly, the fault localization technique based on complex network theory in (Zakari, Lee, & Chong, 2018) will be used for localizing the faults in each *fault-focused* community.

3.6. Chapter Summary

This chapter has discussed the general methodology of this research. The methodology of the investigative study on the claimed problematic parallel debugging approach was presented in detail. The chapter also described the proposed multiple fault localization technique based on complex network theory (FLCN-M). Furthermore, the single fault localization technique based on complex network theory (FLCN-S) was also presented. Lastly, the chapter concluded by presenting the new community-based fault isolation approach. The next chapter presents the experimental setups carried out for the implementation and validation of the investigative study of the claimed problematic parallel debugging approach, the two proposed techniques, and the proposed approach.

CHAPTER 4: EXPERIMENTATION

In this chapter, the different experimental setups carried out for the implementation and validation of the investigation study of the claimed problematic parallel debugging approach, the two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), and the new community-based fault isolation approach are discussed in detail. The data collection process is also highlighted. The subject programs, the evaluation metrics, the techniques and approaches used for cross-comparisons in each experiment are also presented.

4.1. Data Collection

For all the subject programs used in this thesis, each faulty version was executed with all its available test cases. All the programs were executed on a PC with 2.13 GHz Intel Core 2 Duo CPU and an 8GB physical memory. To compile the programs and obtain the code coverage information for each test case, GCC compiler is used for the former while Gcov is used for the latter. Success and failure of a given test case was determined by the outputs of program faulty version and its correct version. If the output of the faulty version is different from the output of the corresponding correct version, the test case will be recorded as a failed test. However, if the outputs do not differ, the test case will be recorded as a passed test.

Following the documentation and the experimental process in previous studies (Abreu et al., 2009b; Liblit et al., 2005; X. Wang & Liu, 2016; Z. Zhang, Chan, & Tse, 2012; Z. Zhang, Jiang, Chan, Tse, & Wang, 2010), versions whose faults cannot be revealed by any test cases were excluded, because any faulty version that did not reveal any failure of at

least a single test case will not be useful to the experiments, mainly because the fault localization metrics used for cross-comparison and the proposed techniques and approach require both passed and failed tests execution for fault diagnosis. The data collected (tests execution/tests result) are known as program spectra. The data collection process is performed on all the experiments in this thesis.

4.2. Experiments

In this section, the different experiments carried out in this research are presented. Four experiments were conducted in this thesis, each composed of the subject programs, the evaluation metrics, and the techniques or approaches used for cross-comparison. For the experiments, data collections are done based on the process detailed in Section 4.1. However, for experiment 2, experiment 3, and experiment 4, the data collected is modeled based on the process highlighted in Chapter 3, Section 3.3.1.

4.2.1. Experiment 1: Investigative study of the Claimed Problematic Parallel Debugging Approach

This experiment primarily aims to investigate the claimed problematic parallel debugging approach in terms of localization effectiveness in comparison with two other debugging approaches. Firstly, an investigative study is conducted on the usefulness of the claimed problematic parallel debugging approach that makes use of *k*-means clustering algorithm (that groups tests execution based on their execution profile similarity) with Euclidian distance metric on three well-known similarity coefficient-based fault localization techniques (Ochiai, Naish2, and Jaccard). Secondly, a cross-comparison is conducted between the claimed problematic parallel debugging approach and OBA

debugging approach in terms of localization effectiveness. Additionally, a comparative study is conducted between the claimed problematic parallel debugging approach and MSeer parallel debugging approach proposed by Gao et al. (R. Gao & Wong, 2017). The methodology of this experiment was discussed in detail in Chapter 3, Section 3.2.

4.2.1.1. Subject Programs

The experiment is conducted on six subject programs ranging from medium-sized to large-sized programs, namely *tcas, replace, gzip, sed, flex*, and *grep*. These programs are relative in size ranging from 173 to 13,892 lines of code, to help in the rigorous evaluation of the experiment. These programs were also used in many previous studies for fault localization (Abreu et al., 2011; Pearson et al., 2017; E. Wong et al., 2008; W. E. Wong, Debroy, Golden, et al., 2012; W. E. Wong & Qi, 2009). The programs were all downloaded (including all the faulty versions and their test suites) from the software infrastructure repository (SIR) site (http://sir.unl.edu/portal/index.php) (Do, Elbaum, & Rothermel, 2005). This helps in evaluating the claimed problematic parallel debugging approach.

Table 4.1: Experimental subject programs

Program	Description	Lines of code (LOC)	Faulty versions	Test cases
tcas	Altitude separation	173	41	1608
replace	Pattern replacement	564	32	5542
gzip	Data compression	6573	5	211

Table 4.1, continued

sed	Textual manipulator	12062	7	360
flex	Lexical analyser	13,892	22	525
grep	Pattern searcher	12,653	7	470

All the programs are written in C programming language. Table 4.1 shows the details of all the programs: program name, program description, faulty versions, number of lines of code, and number of test cases.

The *tcas* program is an aircraft collision avoidance system that takes twelve numbers as an input which represents distinct flight parameters of two aircrafts and generates resolution advisory as output. Therefore, the output can be *unresolved*, *upward*, and *downward*.

The *replace* program takes three inputs which are *pattern*, *substitute*, and *input text*. The program finds every match of a pattern in the *input text* and replaces it with *substitute*. A *pattern* is a restricted form of regular expression while a *substitute* is a string that allows three meta-characters to be used. For example, if the string that matches *pattern* is *ab* and *substitute* is *a&c*, therefore all the occurrences of *ab* in the input file are replaced with *aabc*.

The *gzip* program is utilized for file compression and decompression. The program is commonly used to reduce the size of name files. The input of *gzip* program includes 13 options and a list of files. For instance, "-S" option is used to define the suffix of the result file, where the default is ".gz".

The *sed* program reads and performs basic transformations on an input stream. It is basically used to parse textual input and apply a user-specified transformation to the input. The program takes as input a *sed* script and one or more text files. For the script file, it includes some *sed* commands such as *append*, *replace*, *delete*, and *insert*. Additionally, options are available to control the behavior of the *sed* program. For instance, "-r" option

is used to have lengthy regular expressions in the script rather than basic regular expressions.

The *flex* program is a lexical analyzer or scanner generator. The program reads a given input file (or files) and generates a C source file, called scanner. The input files contains pairs of regular expression and C code, called rules. Furthermore, there are numerous options to control the behavior of *flex* program. For instance, "-d" is to allow debugging mode in the scanner.

The *grep* program has two input parameters which are *patterns* and *files*. The program prints lines in each file that contains a match of any of the patterns. Therefore, to control the behavior of the program, different options can be utilized. For instance, "-w" causes the program to print only lines containing whole-word matches.

In addition to the existing faulty versions of the above programs, more faulty versions were created using mutation-based fault injection technique (DiGiuseppe & Jones, 2015; R. Gao & Wong, 2017). Existing studies have shown that mutation-based faults can be useful to represent real faults and provide reliable results in program debugging experiments (Andrews, Briand, & Labiche, 2005; Andrews, Briand, Labiche, & Namin, 2006). More faults are generated through arithmetic replacement, increment and decrement of data variables, assignment operator by another operator from the same class, rational/logical error or decision negation in an *if* or *while* statement. By seeding faults from single-fault versions into the multiple-fault versions, faulty versions with multiple faults were generated with 2, 3, 4, and 5 faults for each program. This method of multiple faults generation has been utilized by various studies (Abreu et al., 2011; Högerle et al., 2014; Huang et al., 2013; W. E. Wong, Debroy, Golden, et al., 2012). Overall, 540 multiple-fault versions were generated with 2, 3, 4, and 5 faulty versions for this experiment.

4.2.1.2. Evaluation Metrics

To measure the effectiveness of a given fault localization technique especially in the context of multiple faults, suitable metrics are essential for evaluation. In this experiment, three metrics were utilized, namely the average number of statements examined, the total developer expense (TDE), and Wilcoxon signed-rank test.

(a) ***Average number of statements examined***

Using this metric, the average number of statements that a developer need to examine to find faults in a subject program with multiple faults will be computed (W. E. Wong., 2014). Suppose there is a program P with *n* multiple-fault versions where $X(i)$ and $Y(i)$ are the number of statements that need to be examined to locate all the faults in the *i*th multiple-faulty version by two debugging approaches *X* and *Y*, respectively. Approach *X* is more effective than approach *Y* if approach *X* requires a developer to examine less amount of statements than approach *Y* to find all faults in the faulty versions as shown in Equation 4.1.

$$\frac{\sum_{i=1}^{n} X(i)}{n} < \frac{\sum_{i=1}^{n} Y(i)}{n} \qquad (4.1)$$

(b) ***Total developer expense (TDE)***

A metric named "*Score"* originally proposed by Renieris et al. has been used by various studies to evaluate their proposed fault localization techniques (Renieres & Reiss, 2003). This metric is used to measure a developer effort in locating a single fault in a program. The metric, *Score*, is defined as the percentage of code that need *not* be examined

to find a fault (Equation 4.2), whereby "*rank of fault"* represents the location or rank where the statement containing the fault is identified.

$$Score = \left(1 - \frac{\text{rank of fault}}{\text{Number of executable statements}}\right) \times 100\% \quad (4.2)$$

However, for evaluation in this experiment, instead of using the percentage of program code that need *not* be examined to find faults, a variation of this metric, named EXAM Score or sometimes called *Expense* Score is used. EXAM Score is defined as the percentage of code that a developer has to examine until the first fault is located. This metric is computed by Equation 4.3.

$$\text{EXAM Score} = \frac{\text{rank of fault}}{\text{number of executable statements}} \times 100\% \quad (4.3)$$

For this experiment, the original EXAM Score as presented in Equation 4.3 is extended to a metric named total developer expense (TDE) to evaluate the localization of multiple faults using an OBA debugging approach and a parallel debugging approach (James A Jones et al., 2007).

For OBA debugging approach, if there are *p* debugging iterations, the TDE score will be defined as stated in Equation 4.4.

$$TDE = \sum_{i=1}^{p} \text{EXAM Score } (i) \quad (4.4)$$

where EXAM Score (*i*) is the percentage of program statements that a developer need to examine to locate the first fault at *i*th debugging iteration. For example, suppose there is

a program with two faults and 200 program statements. If a developer has to examine 20 and 8 program statements in the first and second debugging iterations, the EXAM Score for the first iteration will be 20/200*100 = 10 and 8/200*100 = 4 for the second iteration, respectively. Hence, the TDE score for locating all the two faults in this program is 10 + 4 = 14 using the OBA debugging approach. However, for parallel debugging approach, the TDE score will be computed for each *fault-focused* cluster given to a developer. The effort a developer spent in finding a fault for each *fault-focused* cluster can be measured. Using the metric TDE in Equation 4.5, more than one developer can be used to examine the faults unlike using the OBA debugging approach. Let p be the number of debugging iterations and q be the number of *fault-focused* clusters generated in each debugging iteration. Therefore, TDE score can be represented as stated in Equation 4.5.

$$TDE = \sum_{i=1}^{p} \sum_{j=1}^{q} \text{EXAM Score}(i, j) \qquad (4.5)$$

Where EXAM Score (i, j) is the percentage of program statements that is needed to be examined to locate the faults for *j*th *fault-focused* cluster in *i*th debugging iteration. For illustration purpose, suppose there is a program version with five faults and 200 program statements. Suppose that the proposed approach needs three debugging iterations to locate all the faults. For the first debugging iteration, two *fault-focused* clusters are generated for developers to check for the faults. For the first cluster, 5 statements need to be examined to locate the fault (which will be 5/200*100 = 2.5), while 12 statements have to be examined in the second cluster to find another fault (which will be 12/200*100 = 6). In the second debugging iteration, two more *fault-focused* clusters are generated. For the first cluster, the developer needs to examine 6 statements to locate the faults (which will be 6/200*100 = 3), while 7 statements have to be examined in the second cluster (which will be 7/200*100

= 3.5). Furthermore, for the third debugging iteration, there is only one *fault-focused* cluster generated which requires an examination of 10 statements to locate the remaining fault (which will be 10/200*100 = 5). In totality, the TDE score to locate all the five faults will be 2.5 + 6 + 3 + 3.5 + 5 = 20. For a fault localization technique utilizing the OBA debugging approach or parallel debugging approach on multiple-fault subject programs, its effectiveness can be computed using TDE. If technique *X* has a lesser TDE score than technique *Y*, then technique *X* is considered more effective than technique *Y*.

(c) Wilcoxon Signed-Rank Test

Wilcoxon signed-rank test which is also known as Mann-Whitney U test is an alternative option to other existing hypothesis tests such as z-test and paired student's t-test particularly when a normal distribution of a given population cannot be assumed (Ott & Longnecker, 2015; W. E. Wong et al., 2014). Wilcoxon signed-rank test is also utilized to give a comparison with a solid statistical basis between different techniques in terms of effectiveness. After computing the number of statements that a developer needs to examine on all approaches, an evaluation will be conducted on the one-tailed alternative hypothesis that the baseline approaches used for cross-comparison require the examination of an equal or greater number of statements than the proposed approach.

Hence, the null hypothesis, in this case, specifies that the baseline approaches require to examine fewer statements than the proposed approach. The null hypothesis is stated as follows:

H_0: The number of statements examined by the baseline approaches to locate all faults in a multiple-fault program ≤ the number of statements examined by the proposed approach.

Therefore, if H_0 is rejected, the alternative hypothesis is accepted. The alternative hypothesis implies that the proposed approach will require the examination of fewer

statements than the baseline approaches which indicates that the proposed approach is more effective.

4.2.1.3. Approaches for Cross-comparison

For the experiment on the effectiveness of the claimed problematic parallel debugging approach in localizing multiple faults, three well-known similarity coefficient-based fault localization techniques are used. The three similarity coefficient-based fault localization techniques are Ochiai coefficient, Naish2 coefficient, and Jaccard coefficient. Furthermore, for cross-comparison with the claimed problematic parallel debugging approach, two other debugging approaches were considered, namely OBA debugging approach and MSeer parallel debugging approach.

The OBA debugging approach uses Ochiai similarity coefficient-based fault localization technique for fault localization. The second approach used for cross-comparison is MSeer parallel debugging approach (R. Gao & Wong, 2017). MSeer uses an improved *k*-medoids clustering algorithm to perform tests clustering and a revised Kendall tau distance metric to measure the distance between two failed tests with Crosstab fault localization technique for fault localization (W. E. Wong, Debroy, & Xu, 2012).

4.2.2. Experiment 2: Multiple Fault Localization based on Complex Network Theory (FLCN-M)

In this experiment, a technique named multiple fault localization based on complex network theory (FLCN-M) is proposed to localize multiple faults effectively and to aid developers to localize multiple faults simultaneously in a single diagnosis rank list. This

work has been published in (Zakari, Lee, & Chong, 2018). The methodology of this experiment was discussed in detail in Chapter 3, Section 3.3.

4.2.2.1. Subject Programs

Table 4.2 shows the details of the programs used in this experiment. The seven programs in Table 4.2 were first used in the single-fault experiment where each of these programs contains only a single fault. The programs were all downloaded from SIR site (http://sir.unl.edu/portal/index.php).

The two programs, *print_tokens* and *print_tokens2*, are basically used to tokenize input file and define the type of each token. Therefore, a token can be one of the following types: *identifier*, *special*, *keyword*, *number*, *comment*, *character constant* or *string constant*. For *replace* program, the program find every match of pattern in the *input text* and replace it with *substitute* (detailed in Section 4.2.1.1).

The two programs *schedule* and *schedule2*, takes the same input and produce the same output. However, the two programs use distinct scheduling algorithms. The input of these programs includes, first, three non-negative integers representing the number of processes in three different priority queues which are *low*, *medium*, and *high*. Second, a list of commands that has to be executed on queues. These commands are, *new_job*, *upgrade_prio*, *block*, *unblock*, *quantum_expire*, *finish*, and *flush*. The output of the programs is a list of numbers indicating the order in which the processes exit (from the scheduling system).

Furthermore, *tcas* program is an aircraft collision avoidance system (detailed in Section 4.2.1.1). Lastly, *tot_info* program takes a file that contains one or more tables as

input. The program uses the notions of chi-square and degree of freedom to calculate whether the distribution of numbers in the tables is logarithm gamma distribution. Hence, the output is the total degree of freedom of rows and columns and chi-square.

Table 4.2: Summary of Siemens test suite programs

Program	Faulty version	Lines of code (LOC)	Test cases
print_tokens	7	565	4130
print_tokens2	10	510	4115
replace	32	412	2650
schedule	9	307	2710
schedule2	10	563	5542
tcas	41	173	1608
tot_info	23	406	1052

In order to evaluate the proposed technique (FLCN-M) with multiple faults, the multiple-fault versions of Siemens test suite are adopted as used in (Abreu et al., 2011). Five out of the seven Siemens test suite programs in Table 4.2 were used, namely *tcas, print_tokens, print_tokens2, replace,* and *schedule*. Several faults from their former versions were combined and manually seeded into the associated programs to create faulty program versions with 2, 3, 4, and 5 faults each. These faulty versions are named as SIEMENS-M. This will help in the rigorous evaluation of the proposed fault localization technique (FLCN-M) on programs with multiple faults. To further help in result generalization, UNIX real-life utility programs were utilized, which are *gzip* and *sed* as presented in Table 4.3. These programs contain both real and seeded faults, and they are relatively larger-sized programs compared with Siemens test suite programs.

These programs were also downloaded from SIR, and they were all written in C programming language.

Table 4.3: UNIX real-life utility programs

Program	Faulty versions	Test cases	Lines of code (LOC)	Description
gzip	5	211	6573	Data compression
sed	7	360	12062	Textual manipulator

4.2.2.2. Evaluation Metrics

Two metrics are used in this experiment, which are EXAM Score and Incremental Developer Expense (IDE).

(a) ***EXAM Score***

EXAM Score is defined as the percentage of code that a developer has to examine to find a fault. It can also be defined as the percentage of code that needs to be examined until the first statement where the fault resides is reached. The metric is depicted in Equation 4.3.

This metric will be specifically used in programs that contain only a single fault (Siemens test suite) in this experiment. Generally, for any fault localization technique, its effectiveness can be accessed and compared with EXAM Score, whereby if technique A has a lesser EXAM Score than technique B, then technique A will be considered to be more effective because less code is needed to be examined to locate the faults.

(b) ***Incremental Developer Expense (IDE)***

For multiple-fault programs, a different metric is proposed called the Incremental Developer Expense (IDE) to aid in accessing a technique's effectiveness on localizing multiple faults in a single diagnosis rank list. For fault *i,* IDE is formulated as shown in Equation 4.6.

$$IDE = \text{EXAM Score } (1) + \sum_{i=2}^{n} (\text{EXAM Score } (i) - \text{EXAM Score } (i-1)) \qquad (4.6)$$

where *n* represents the number of faults in a faulty program and EXAM Score (*i*) is the total effort a developer needed to locate a fault. The main objective of this metric is to allow the developer to continue searching for faults in a single diagnosis rank list until all the program faults are located. The fault localization process will not be interrupted even if the first fault is found. The process will halt when the maximum number of faults is located. To elaborate more on the working of the evaluation metric, the following example is given. For example, suppose there is a program with three faults and 12 program statements where statement 3, statement 5, and statement 8 contain the faults. If a single faulty diagnosis is obtained as D = {8, 3, 2, 5, 10, 12, 6, 7, 4, 1, 11, and 9} based on IDE, for the first fault located in statement 8, this diagnosis is said to have an expense of 1/12*100 = 8.3 (EXAM Score (1)).

For the second fault located in statement 3, the expense will be calculated as 2/12*100 – 8.3 = 8.4 (EXAM Score (2) − EXAM Score (1)). For the third fault in statement 5, the expense will be calculated as 4/12*100 – 16.7 = 16.6 (EXAM Score (3) − EXAM Score (2) − EXAM Score (1)). Therefore, the IDE to locate all the three faults in a single diagnosis rank list for the above example will be sum up as 8.3 + 8.4 + 16.6 =

33.3 as presented in Equation 4.6. This metric calculates IDE based on how many program statements a developer has to check to find the next fault. Normally, a developer does not know how many faults exist in a program when the program fails. As a stopping criterion, a developer is required to stop the fault localization process if he/she searches about 70% of the program executable code. If the debugging effort has reached the assigned stipulated percentage (70%), then the process will be stopped. The program will be re-tested, and if any test case fails, the debugging process will start all over again until the program is fault-free.

4.2.2.3. Techniques for Cross-comparison

To evaluate the proposed multiple fault localization technique (FLCN-M), the technique is first evaluated on single-fault subject programs. Therefore, the following fault localization techniques which are generally known as some of the best techniques on single fault context, namely Tarantula (J. A. Jones & Harrold, 2005), Sober (Liblit et al., 2005), Delta Debugging (DD) (Zeller, 2002), Nearest Neighbour (NN) (Renieres & Reiss, 2003), SNCM (Zhu et al., 2011), Intersection and Union (Renieres & Reiss, 2003), are plotted for the comparative analysis for single-fault programs' evaluation.

Furthermore, three fault localization techniques were used for cross-comparison on multiple-fault subject programs. Out of which two are similarity coefficient-based fault localization techniques, namely Ochiai coefficient and Tarantula coefficient, and an existing fault localization technique based on software network centrality measures (SNCM). The similarity coefficient-based fault localization techniques (Ochiai coefficient and Tarantula coefficient) were used in localizing multiple faults in few studies (Abreu et al., 2011; James A Jones et al., 2007; W. E. Wong, Debroy, Golden, et al., 2012). SNCM was never utilized on multiple-fault programs. However, it is used for cross-comparison

because as far as the knowledge gained from the literature, it is the only fault localization technique that makes use of centrality measures for fault diagnosis. Therefore, it is important to compare the proposed technique with SNCM in this experiment.

4.2.3. Experiment 3: Single Fault Localization based on Complex Network Theory (FLCN-S)

In this experiment, a technique named single fault localization based on complex network theory (FLCN-S) is proposed. FLCN-S technique is proposed to improve localization effectiveness on single-fault context. The methodology of this experiment was discussed in detail in Chapter 3, Section 3.4.

4.2.3.1. Subject Programs

For this experiment, the seven Siemens test suite subject programs as shown in Table 4.2 (Hutchins, Foster, Goradia, & Ostrand, 1994) and two UNIX real-life utility programs as shown in Table 4.3 (Do et al., 2005) were utilized to evaluate the proposed fault localization technique (FLCN-S). All of these subject programs are written in C programming language. Siemens test suite is composed of seven subject programs, namely *schedule, schedule2, print_tokens, print_tokens2, replace, tot_info,* and *tcas*. Each of the subject programs has more than 1000 test cases.

4.2.3.2. Evaluation Metrics

In this experiment, three metrics were utilized, namely the cumulative number of statements examined, the total developer expense (TDE), and Wilcoxon signed-rank test.

(a) ***EXAM Score***

To access the overall effectiveness of a fault localization technique, a suitable metric must be used for evaluation. In this experiment, the metric named EXAM Score is utilized which is depicted in Equation 4.3.

Many studies have used EXAM Score to access the effectiveness of a single fault localization technique (DiGiuseppe & Jones, 2015; E. Wong et al., 2008; W. E. Wong et al., 2014; W. E. Wong, Debroy, Golden, et al., 2012; W. E. Wong & Qi, 2009).

(b) ***Cumulative number of statements examined***

In addition to using EXAM Score, the cumulative (total) number of statements that need to be examined with respect to faulty versions of a subject program to locate faults is also considered (W. E. Wong., 2014). Therefore, for n faulty versions of a given subject program P where $X(i)$ and $Y(i)$ are the number of statements that need to be examined to locate all the faults in the ith faulty version by techniques X and Y, respectively. Technique X is more effective than technique Y if technique X requires a developer to examine a lesser number of statements than technique Y to find all faults in the faulty versions as shown in Equation 4.10.

$$\sum_{i=1}^{n} X(i) < \sum_{i=1}^{n} Y(i) \qquad (4.10)$$

(c) ***Wilcoxon Signed-Rank Test***

To evaluate the proposed technique with sound statistics, Wilcoxon signed-rank test is used. Since the aim is to show that the proposed technique is more effective than the baseline fault localization techniques, the one-tailed alternative hypothesis that the baseline techniques are required to examine the same or a greater number of statements is evaluated. The null hypothesis is stated as follows:

H_0: The number of statements examined by the baseline techniques ≤ the number of statements examined by the proposed technique.

Therefore, if H_0 is rejected, the alternative hypothesis is accepted.

4.2.3.3. Techniques for Cross-comparison

To evaluate the proposed single fault localization technique (FLCN-S) on single-fault subject programs, two similarity coefficient-based fault localization techniques which are Ochiai coefficient and Jaccard coefficient were utilised for cross-comparison.

4.2.4. Experiment 4: Community-based Fault Isolation Approach

In this experiment, a new community-based fault isolation approach is proposed to aid in the effective isolation and localization of multiple faults simultaneously in parallel. The methodology of this experiment was discussed in detail in Chapter 3, Section 3.5.

4.2.4.1. Subject Programs

For this experiment, six subject programs were used ranging from medium-sized to large-sized programs, namely *tcas, replace, gzip, sed, flex*, and *grep* as utilized in Experiment 1, Table 4.1. For multiple-fault versions generation, mutation-based fault injection technique was utilized as explained in Section 4.2.1.1 (DiGiuseppe & Jones, 2015).

4.2.4.2. Evaluation Metrics

In this experiment, three evaluation metrics were utilized which are stated as follows:

- Average number of statements examined.
- Total developer expense (TDE).
- Wilcoxon signed-rank test.

These evaluation metrics are also used for the investigative study in Section 4.2.1 and detailed in Section 4.2.1.2.

4.2.4.3. Approaches for Cross-comparison

The community-based fault isolation approach is compared with two baseline approaches. The first approach uses the same parallel debugging process as used by Jones et al. (James A Jones et al., 2007) with Ochiai coefficient metric. Furthermore, the approach is the same as the claimed problematic parallel debugging approach investigated in Experiment 1, Section 4.2.1. Henceforth, the approach is referred to as P-Ochiai. P-Ochiai applies a *k*-means clustering algorithm to cluster failed tests execution with Euclidian distance metric to measure the distance between failed tests as used in (Huang et al., 2013).

On the other hand, the second approach used for cross-comparison is MSeer parallel debugging approach (R. Gao & Wong, 2017). MSeer uses an improved *k*-medoids clustering algorithm to perform tests clustering with a revised Kendall tau distance metric to measure the distance between two failed tests. Both approaches measure the *due-to* relationship between failed test cases to create *fault-focused* clusters. In contrast, the newly proposed community-based fault isolation approach measures the *due-to* relationship between program statements based on edge-betweenness distance to create *fault-focused* communities. For P-Ochiai, suspiciousness rankings are generated using Ochiai coefficient, while MSeer uses Crosstab fault localization technique (W. E. Wong, Debroy, & Xu, 2012).

4.3. Chapter Summary

This chapter has discussed four experiments that were carried out in this thesis. Firstly, the data collection process was presented. Secondly, the subject programs, the evaluation metrics, and the techniques and approaches used for cross-comparisons for

each experiment were presented in detail. Finally, the chapter concludes the discussion of the experimental setups. The next chapter presents the results and discussion of each experiment in this research with the cross-comparisons between all the baseline techniques and approaches.

CHAPTER 5: RESULTS AND DISCUSSION

In this chapter, the different experimental results that were carried out for the investigative study of the claimed problematic parallel debugging approach, the two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), as well as the new community-based fault isolation approach, are presented and discussed in detail. The cross-comparisons with the existing baseline techniques and approaches in the field of research are also presented.

5.1. Experiment 1

This section discusses the experimental results of the investigative study of the claimed problematic parallel debugging approach presented in Chapter 4, Section 4.2.1. All the experiments in this section were evaluated based on three metrics, namely the average number of statements examined, TDE score, and Wilcoxon signed-rank test. For all the experiments, the result of a given approach is categorized as best or worst case based on its performance on how many statements a developer needs to check to find the faulty statement. For clarity, consider that there are two faulty statements in a given program's statements rank list. In the best case, the faulty statement is examined first, or at most the statement resided at the top-most of the ranking list; while in the worst case, the developer has to examine the faulty statement last whereby he/she has to examine many correct statements before the faulty statement is located. In other words, the faulty statement resides at the lowest end of the rank list.

5.1.1. The Effectiveness of the Claimed Problematic Parallel Debugging Approach

In this section, the results for the investigative study on the effectiveness of the claimed problematic parallel debugging approach on three well-known similarity coefficient-based fault localization techniques (Ochiai coefficient, Naish2 coefficient, and Jaccard coefficient) is presented. The claimed problematic parallel debugging approach adopt the three coefficients as P-Ochiai, P-Naish2, and P-Jaccard in its fault localization process in the experiment.

(d) ***Average number of statements examined***

Table 5.1 highlights the average number of statements that are needed to be examined by each approach to find all the faults. Each subject program contains *x* number of faults (*x* = 2, 3, 4, and 5). For this experiment, only the best cases for each subject program's versions are plotted because presenting the worst case is not necessary. However, it is worth knowing that these values correspond to the average number of statements that each approach requires to examine to locate all faults in the respective programs' faulty versions. Table 5.1 presents the average number of statements examined by P-Ochiai, P-Naish2, and P-Jaccard with respect to 25 versions of each program containing *x* amount of faults (*x* = 2, 3, 4, and 5).

For instance, it was observed that, for *replace* program with respect to 3-fault faulty versions, the average number of statements to be examined to locate all the faults by P-Ochiai, P-Naish2, and P-Jaccard are 24.05, 51.29, and 60.13, respectively. It was observed that on 4 and 5 faulty versions, P-Jaccard is more effective than P-Naish2. For example, in the 4-fault versions of *tcas*, *replace*, *gzip*, *sed, flex,* and *grep* programs, using P-Jaccard, the average number of statements examined to find all the faults are 34.89

(*tcas*), 82.98 (*replace*), 180.11 (*gzip*), 530.22 (*sed*), 189.33 (*flex*), and 614.11 (*grep*). In contrast, using P-Naish2, the average statements examined are 35.16 (*tcas*), 85.09 (*replace*), 183.18 (*gzip*), 525.11 (*sed*), 202.10 (*flex*), and 645.00 (*grep*), respectively. In these faulty versions (4-fault versions), it shows that P-Jaccard is on average 2% more effective than P-Naish2. The most significant point worth noting is that P-Ochiai is consistently more effective than the rest of the approaches (P-Naish2 and P-Jaccard). However, it is not surprising because Ochiai coefficient in earlier studies was found to be the most effective coefficient in fault localization inferencing (Abreu et al., 2007).

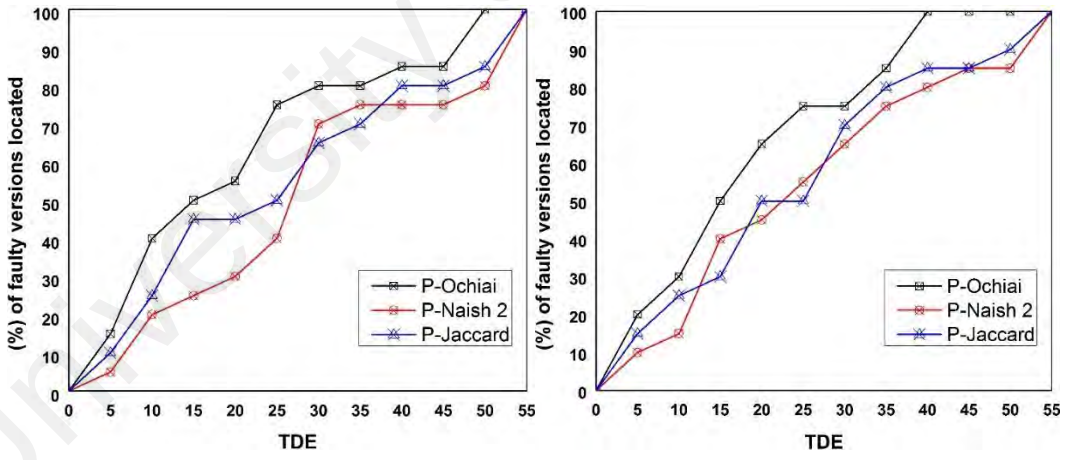
Table 5.1: Average number of statements examined (best case)

		tcas	replace	gzip	sed	flex	grep
2-fault	P-Ochiai	10.09	19.14	40.03	81.48	35.23	370.01
	P-Naish2	25.40	29.16	45.53	75.03	37.28	379.14
	P-Jaccard	24.90	30.00	60.01	82.17	41.83	383.84
3-fault	P-Ochiai	22.07	24.05	80.63	393.89	110.03	512.15
	P-Naish2	29.49	51.29	87.16	414.01	120.83	525.49
	P-Jaccard	28.02	60.13	95.19	412.03	119.16	545.69
4-fault	P-Ochiai	33.33	60.04	161.04	500.10	190.00	601.60
	P-Naish2	35.16	85.09	183.18	525.11	202.10	645.00
	P-Jaccard	34.89	82.98	180.11	530.22	189.33	614.11
5-fault	P-Ochiai	49.04	98.91	260.03	679.74	230.06	621.34
	P-Naish2	53.08	120.01	285.01	699.11	260.11	640.33
	P-Jaccard	51.89	112.03	275.33	711.08	245.03	641.44

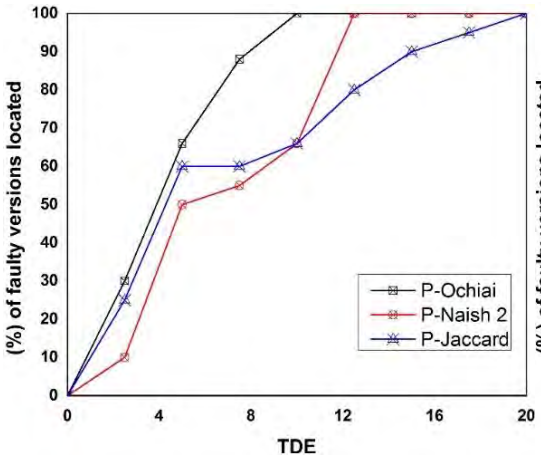
(e) Total developer expense (TDE)

Furthermore, the evaluation of the claimed problematic parallel debugging approaches (P-Ochiai, P-Naish2, and P-Jaccard) based on TDE score is presented. The 2-fault versions of six programs (*tcas, replace, gzip, sed, flex,* and *grep*) are plotted in Figure 5.1 with respect to all faulty versions. In this figure, the *y*-axis represents the percentage

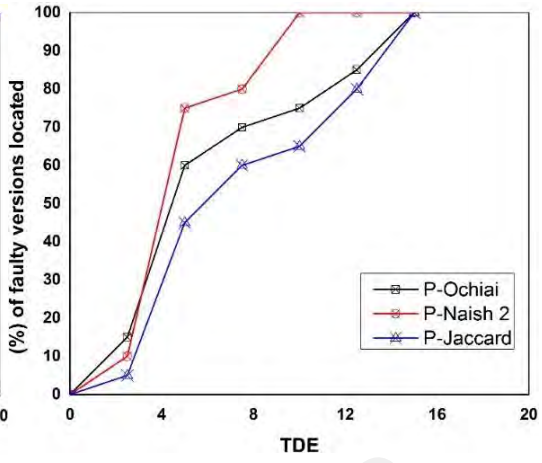
(%) of faulty versions located by a developer while the *x*-axis represents the percentage of code (total developer expense) examined. Based on part (a) of Figure 5.1 (*tcas* 2-fault versions), it was observed that by examining less than 25% of the program code, developers on P-Ochiai can locate 75% of the faulty versions, while on P-Jaccard and P-Naish2, the developers can only locate 50% and 40% by examining the same amount of program code respectively. In part (b) *replace* (2-fault versions), by examining less than 10% of the code, the developers can locate 30% of the faults in the faulty versions using the P-Ochiai approach, and can locate 25% and 15% of the faults using the P-Jaccard and P-Naish2, respectively. The curve in part (d) *sed* (2-fault versions), shows that P-Naish2 records the top performance in locating all the faulty versions by checking less than 10% of the program's faulty versions.



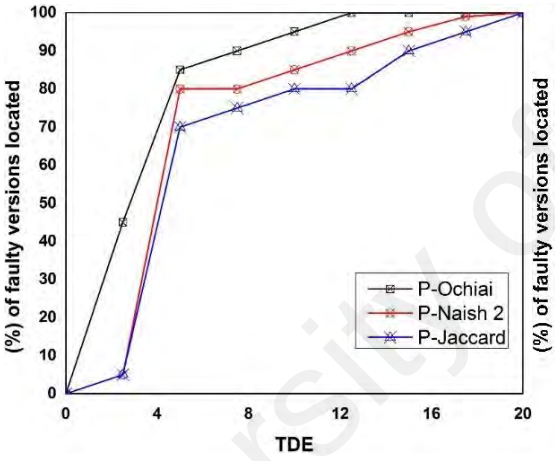
(a) total expense of tcas 2-fault versions (b) total expense of replace 2-fault versions



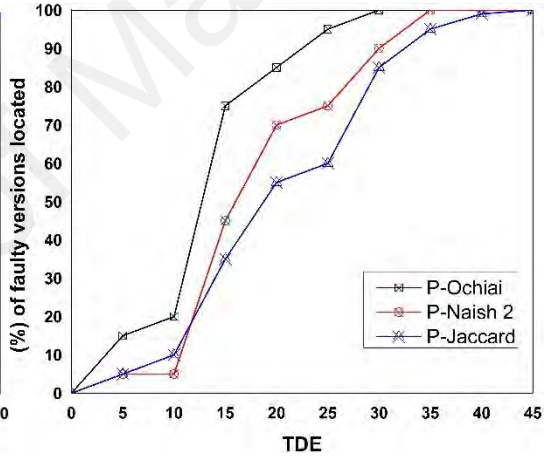
(c) total expense of gzip 2-fault versions



(d) total expense of sed 2-fault versions



(e) total expense of flex 2-fault versions



(f) total expense of grep 2-fault versions

Figure 5.1: TDE score-based comparison of the claimed problematic parallel debugging approach with respect to the three coefficients (best case)

However, looking at part (c) and part (e), P-Ochiai has surpassed the other two legends (P-Jaccard and P-Naish2) in terms of effectiveness. In part (f), *grep* (2-fault versions), by examining less than 20% of the code, the developers can locate 85% of the faults in the faulty versions using the P-Ochiai approach, and can locate 70% and 55% of the faults using P-Naish2 and P-Jaccard, respectively. Table 5.2 presents the effectiveness comparison with which it can be claimed that P-Ochiai is more effective than the other

two approaches (P-Naish2 and P-Jaccard) using the Wilcoxon signed-rank test. The data is presented for P-Ochiai against the corresponding approaches because from Table 5.1, P-Ochiai has consistently shown to be more effective in comparison to P-Naish2 and P-Jaccard. The entries in the table are the confidence with which the alternative hypothesis (which implies that P-Ochiai requires the examination of fewer statements than the compared approaches) can be accepted with respect to a given program in its best case scenario.

For example, one can say with 97.70% confidence that P-Ochiai is more effective than P-Naish2 on *grep* program (4-fault versions). However, for *replace* program on all faulty versions (2-fault, 3-fault, 4-fault, and 5-fault), the confidence to accept the alternative hypothesis is all higher than 90%. In few scenarios, the confidence to accept the alternative hypothesis is lower than 60%, such as P-Ochiai being more effective than P-Naish2 and P-Jaccard with 45.36% and 35.90% confidence for the 4-fault versions of *tcas*, and with 51.22% being more effective than P-Naish2 for the 2-fault versions of *flex*.

Table 5.2: The confidence with which it can be claimed that P-Ochiai is more effective than P-Naish2 and P-Jaccard (best cases)

		tcas	replace	gzip	sed	flex	grep
2-fault	P-Naish2	93.47%	90.02%	81.82%	00.00%	51.22%	89.05%
	P-Jaccard	93.25%	90.80%	95.00%	00.00%	84.85%	92.77%
3-fault	P-Naish2	86.53%	96.33%	84.69%	95.03%	98.54%	92.51%
	P-Jaccard	83.20%	97.23%	93.14%	94.49%	89.05%	97.02%
4-fault	P-Naish2	45.36%	96.01%	95.49%	96.01%	91.74%	97.70%
	P-Jaccard	35.90%	95.65%	94.76%	96.68%	00.00%	92.01%
5-fault	P-Naish2	75.25%	95.27%	96.00%	94.84%	96.68%	94.74%
	P-Jaccard	64.92%	92.38%	93.47%	96.81%	93.32%	95.03%

Out of all the programs, only 3 scenarios the H_0 (null hypothesis) is accepted. For clarification, the cells with a black background in Table 5.2 are the cells where the null hypothesis is accepted and P-Ochiai is outperformed. The null hypothesis is accepted if the number of statements examined by the compared approach is less than that of P-Ochiai, thereby making the compared approach more effective. Therefore, if the null hypothesis is accepted, the confidence level is given as 00.00% in Table 5.2. For example, for *sed* (2-fault versions) of P-Naish2 and P-Jaccard, the confidence to accept the alternative hypothesis is 00.00%, a similar observation is also made for *flex* (4-fault versions) of P-Jaccard.

Conclusively, the results from the Wilcoxon signed-rank test also shows that P-Ochiai is statistically by-large more effective than P-Naish2 and P-Jaccard with the exception in few scenarios. The result is also consistent with the conclusion based on the two other evaluation metrics, namely the average number of statements examined and TDE score in Table 5.1 and Figure 5.1. Based on the result obtained in this section, one can say that the claimed problematic parallel debugging approach is relatively useful and effective in locating multiple faults based on these scenarios. However, to further substantiate this claim, further comparisons with OBA and MSeer debugging approaches is needed. In this regard, P-Ochiai will be used for further comparisons.

5.1.2. Cross-Comparison with OBA Debugging Approach

In this section, P-Ochiai (the best of the claimed problematic parallel debugging approaches) is compared with OBA debugging approach. In this experiment, OBA used the Ochiai coefficient in its fault localization process.

(f) Average number of statements examined

Table 5.3 and Table 5.4 present the average number of statements examined by both P-Ochiai and OBA with respect to 25 versions of each program containing *x* number of faults (*x* = 2, 3, 4, and 5) for both the best and worst cases. It was observed that, for *gzip* program (3-fault versions), P-Ochiai can locate all the faults by examining at least 80.63 statements in the best case and 254.19 in the worst case. However, for OBA, the average number of statements examined in the best case is 91.16 while it is 293.28 in the worst case. For *tcas* 2-fault versions, OBA is more effective where the average number of statements examined is 8.69 for the best case. It was observed that in most cases, the effectiveness of P-Ochiai and OBA is marginal whereby the differences are minor for both the best cases and worst cases. Generally, in terms of the average number of statements examined to localize all faults, P-Ochiai is more effective.

Table 5.3: Average number of statements examined (best case)

		tcas	replace	gzip	sed	flex	grep
2-fault	P-Ochiai	10.09	19.14	40.03	81.48	35.23	370.01
	OBA	8.69	20.18	46.38	85.00	38.02	383.88
3-fault	P-Ochiai	22.07	24.05	80.63	393.89	110.03	512.15
	OBA	25.89	30.15	91.16	380.53	115.50	529.93
4-fault	P-Ochiai	33.33	60.04	161.04	500.10	190.00	601.60
	OBA	40.15	58.33	172.03	485.01	220.11	614.18
5-fault	P-Ochiai	49.04	98.91	260.03	679.74	230.06	621.34
	OBA	47.26	108.11	275.18	640.89	241.67	659.29

Table 5.4: Average number of statements examined (worst case)

		tcas	replace	gzip	sed	flex	grep
2-fault	P-Ochiai	17.49	46.41	194.33	292.09	161.08	1310.01
	OBA	18.01	39.01	167.11	309.01	214.78	1345.11
3-fault	P-Ochiai	28.49	55.19	254.19	601.75	243.63	1901.77

Table 5.4, continued

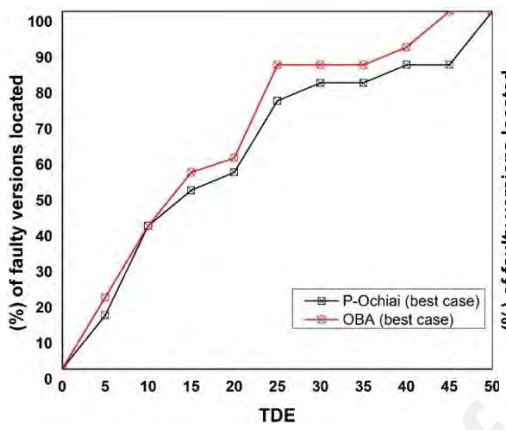
	OBA	27.01	73.16	293.28	593.01	301.36	1983.03
4-fault	P-Ochiai	42.89	100.89	499.01	889.05	294.00	2300.03
	OBA	43.44	114.00	515.11	855.14	340.10	2214.28
5-fault	P-Ochiai	56.02	155.97	570.35	1501.08	322.15	2609.10
	OBA	57.49	170.13	608.37	1424.00	410.80	2711.35

(g) ***Total developer expense (TDE)***

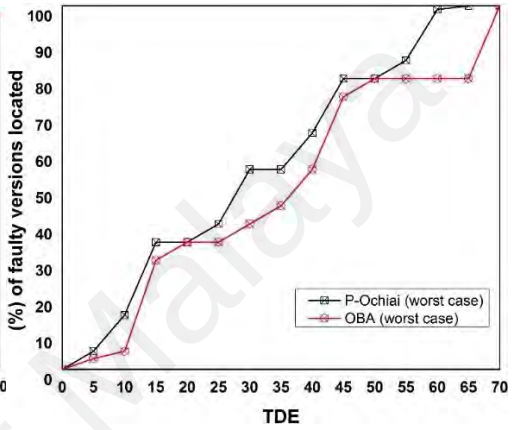
Now, the result of the comparative evaluation between P-Ochiai and OBA with respect to TDE score is presented. In Figure 5.2, the best and worst cases of *tcas, replace,* and *gzip* 2-fault versions are presented on all the faulty versions. In part (a) and part (b) of Figure 5.2, by examining less than 10% of the program code, P-Ochiai can locate 40% of the faulty versions in the best case and 15% in the worst case. In contrast, by examining the same amount of code, OBA can locate 40% and 5% of the faulty versions in the best and worst cases respectively. Moreover, in part (a), OBA was able to locate all the faulty versions by examining less than 45% of the program code, while P-Ochiai can locate all by examining less than 50% of the program code. For the best case of *tcas* 2-fault versions, OBA performed relatively better. Furthermore, part (c) and part (d) highlight the TDE score of *replace* program, by examining less than 10% of the program code, P-Ochiai can locate 30% and 15% of the faulty versions in the best case and worst case, while by examining the same amount of code, OBA can locate 25% in the best case and 5% in the worst case.

P-Ochiai is consistently more effective than OBA in part (e) and part (f) of Figure 5.2. The curves show that, in the best case, P-Ochiai can locate all the program faulty versions (2-fault versions) by examining less than 10% of the program code. Correspondingly, OBA can locate all the program faulty versions by examining less than 12.5% of the *gzip* faulty versions. However, in the worst case, by examining less than

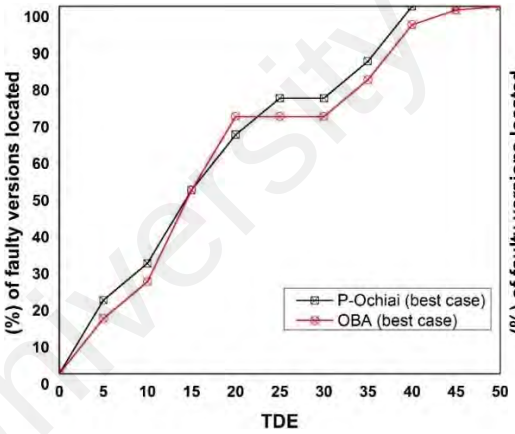
20% of the program code, all the faulty versions can be located using P-Ochiai, and by examining the same amount of code, only 80% of the faulty versions can be located using the OBA debugging approach. In this experiment, the drastic increase in expense with respect to the best case and worst case of each program was observed. In the case of *gzip* in part (e) and part (f), the expense doubled in the worst case for both P-Ochiai and OBA.



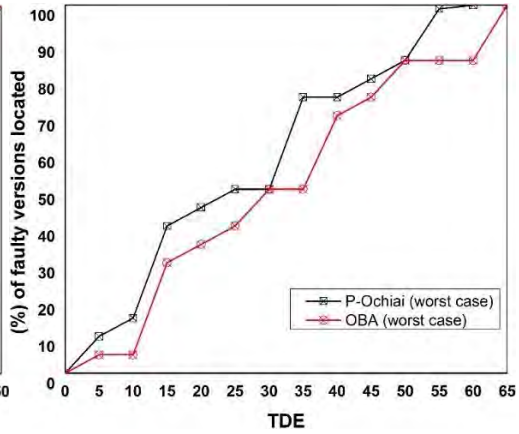
(a) best case of tcas 2-fault versions



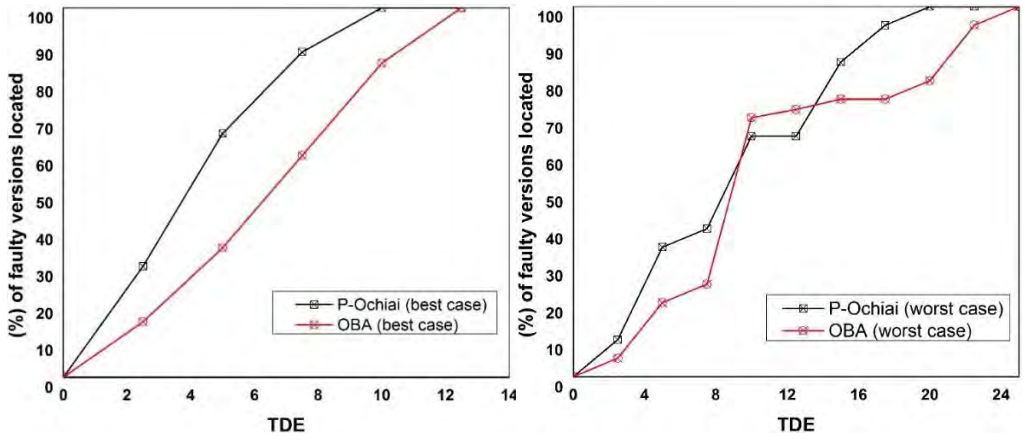
(b) worst case of tcas 2-fault versions



(c) best case of replace 2-fault versions



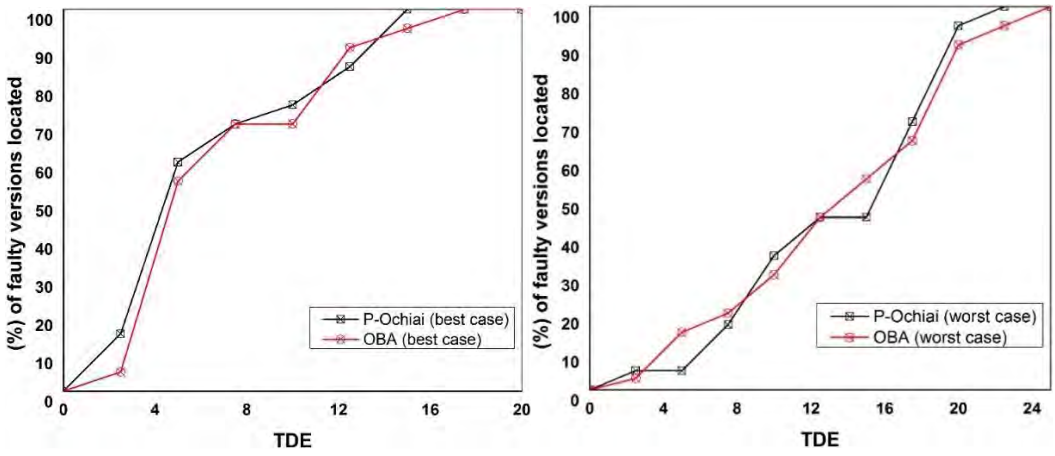
(d) worst case of replace 2-fault versions



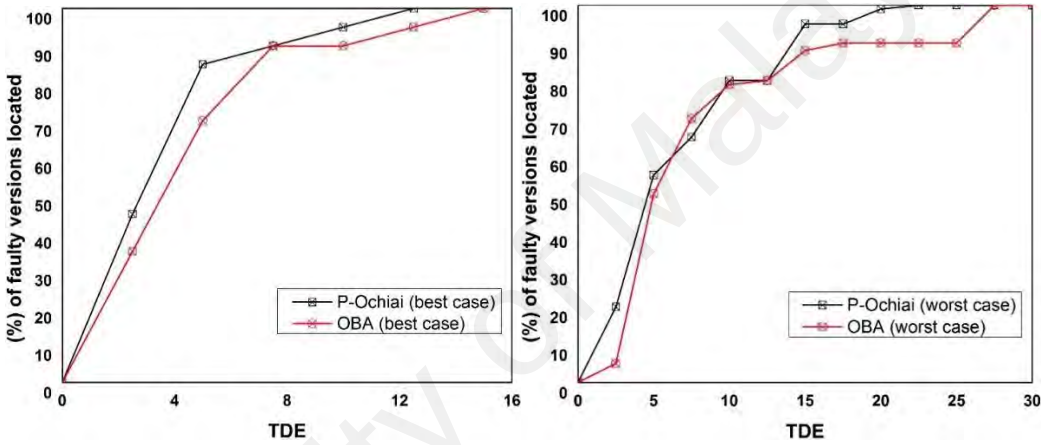
(e) best case of gzip 2-fault versions (f) worst case of gzip 2-fault versions

Figure 5.2: TDE score-based comparison between P-Ochiai and OBA debugging approaches on tcas, replace, and gzip (2-fault versions).

To further give the reader the complete flavor of the result in all programs, Figure 5.3 gives the best and worst case results for *sed* and *flex* programs' faulty versions with 2-fault. For example, in part (a) and part (b) of Figure 5.3, OBA is the second most effective in both the best and worst case scenarios. By examining less than 10% of the program code, the developer using the P-Ochiai can locate 75% and 35% of the faulty versions in both the best and worst cases, respectively. In contrast, OBA can locate 70% and 30% of all the faulty versions in the best and worst cases, respectively. Looking closely at the results, even though the margin between both debugging approaches' effectiveness is minor in the *sed* 2-fault faulty versions, yet the P-Ochiai is relatively persistently more effective. On the other hand, the curves in part (c) and part (d) of *flex* program's faulty versions show that P-Ochiai achieves lower TDE score but still more effective in comparison with OBA.



(a) best case of sed 2-fault versions (b) worst case of sed 2-fault versions



(c) best case of flex 2-fault versions (d) worst case of flex 2-fault versions

Figure 5.3: TDE score-based comparison between P-Ochiai and OBA debugging approaches on sed and flex (2-fault versions).

Table 5.5 and Table 5.6 give the effectiveness comparisons for the best and worst cases using the Wilcoxon signed-rank test. The entries in the tables represent the confidence to accept the alternative hypothesis. For instance, for the *grep* (5-fault versions) worst case scenario, the confidence to accept the alternative hypothesis is higher than 99% (which is the highest in both best and worst cases). In some cases, the confidence to accept the alternative hypothesis is very low. For example, for the best case of *replace* (2-fault versions), the confidence to accept the alternative hypothesis for P-

Ochiai being more effective than OBA is 3.85%, and with 31.98% being more effective than OBA in the 5-fault versions of *tcas*. It was also observed that in a reasonable amount of cases (in both best and worst cases), the null hypothesis is accepted which is represented as having 00.00% confidence in Table 5.5 and Table 5.6. The cells with black background in the tables are the cells where the null hypothesis is accepted and P-Ochiai is outperformed. In these cases, the OBA approach examines a lesser number of statements to find the faults than P-Ochiai, which means it is more effective.

Table 5.5: The confidence with which it can be claimed that P-Ochiai is more effective than OBA approach (best cases)

		tcas	replace	gzip	sed	flex	grep
2-fault	OBA	00.00%	3.85%	84.26%	71.60%	64.16%	92.80%
3-fault	OBA	73.69%	83.61%	90.51%	00.00%	81.72%	94.38%
4-fault	OBA	85.36%	00.00%	90.91%	00.00%	96.68%	92.06%
5-fault	OBA	00.00%	89.14%	93.40%	00.00%	91.39%	97.37%

Table 5.6: The confidence with which it can be claimed that P-Ochiai is more effective than OBA approach (worst cases)

		tcas	replace	gzip	sed	flex	grep
2-fault	OBA	00.00%	00.00%	00.00%	94.09%	98.14%	97.16%
3-fault	OBA	00.00%	94.44%	97.45%	00.00%	98.30%	98.77%
4-fault	OBA	00.00%	92.38%	93.79%	00.00%	97.84%	00.00%
5-fault	OBA	31.98%	92.94%	97.37%	00.00%	98.88%	99.03%

The evaluation does not take into account the time it takes to produces a failure-free program. However, it is a well-known fact that OBA debugging approach often takes a longer time to produce a failure-free program due to the nature of the debugging approach. In OBA, each fault is neutralised per debugging iteration, hence, for a 3-fault program, all the faults will be neutralized in three given iterations (one fault per iteration). In

contrast, for parallel debugging approach, it is expected to localize all the faults simultaneously by generating *fault-focused* clusters which take minimal time and few iterations. In practice, it is difficult to achieve the complete simultaneous identification of software faults in a single debugging iteration whereby in some cases, extra iterations are required to neutralize all the faults in a given faulty program. But overall, with respect to the debugging approach used (P-Ochiai), the time and expense it takes to produce a failure-free program is generally low in comparison with OBA debugging approach.

Another critical observation is that by utilising the claimed problematic parallel debugging approach (P-Ochiai), *fault-focused* clusters are often generated redundantly that target the same fault which reduces the approach's localization effectiveness greatly. This is a further affirmation of the work of Gao et al. (R. Gao & Wong, 2017) where the researchers observed the similar sort of problem when utilizing hierarchical clustering algorithm and Jaccard distance metric for measuring the failed test-to-test distance. In conclusion, one can conclude that based on the experimental tests subjects in this experiment, P-Ochiai is relatively more effective than OBA debugging approach irrespective of the clustering algorithm and distance metric used.

5.1.3. Cross-Comparison with MSeer Debugging Approach

This section provides the results for the cross-comparison between P-Ochiai and MSeer parallel debugging approach.

(h) ***Average number of statements examined***

The results shown in Table 5.7 and Table 5.8 is with respect to 25 versions of each program containing 2, 3, 4, and 5 faults. Table 5.7 and Table 5.8 present the average number of statements that need to be examined by P-Ochiai and MSeer on three subject programs' faulty versions (*gzip, flex,* and *grep*). With respect to *gzip* program (3-fault versions), it was observed that the average number of statements that P-Ochiai examined to locate the faulty versions is 80.63 in the best case and 254.19 in the worst case. For MSeer, 41.77 program statements are examined on average in the best case and 174.57 in the worst case.

Looking at Table 5.7 and Table 5.8, MSeer is by far the best approach. The worst case of MSeer for *flex* 3-fault versions is relatively as effective as the best case of P-Ochiai, whereby the average number of statements examined by MSeer is 111.40 in the worst case, while for P-Ochiai is 110.03 in the best case. Nonetheless, the difference between the two approaches is only 1.37 statements. Henceforth, in most cases, the average number of statements examined by P-Ochiai is 50% to 60% more than the average number of statements examined by MSeer. The conclusion drawn from Table 5.7 and Table 5.8 with respect to the average number of statements examined by both approaches (MSeer and P-Ochiai) is that MSeer is better and more effective than P-Ochiai.

Table 5.7: Average number of statements examined (best case)

		gzip	flex	grep
2-fault	P-Ochiai	40.03	35.23	370.01
	MSeer	12.80	11.37	345.77
3-fault	P-Ochiai	80.63	110.03	512.15
	MSeer	41.77	23.30	492.93
4-fault	P-Ochiai	161.04	190.00	601.60

Table 5.7, continued

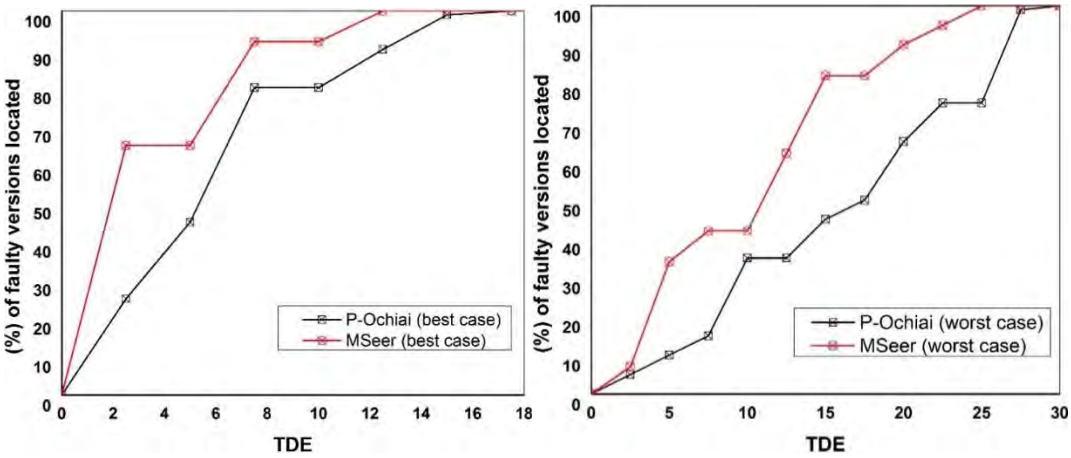
	MSeer	70.27	67.70	560.03
5-fault	P-Ochiai	260.03	230.06	621.34
	MSeer	80.00	104.67	598.80

Table 5.8: Average number of statements examined (worst case)

		gzip	flex	grep
2-fault	P-Ochiai	194.33	162.08	1310.01
	MSeer	77.33	75.87	650.67
3-fault	P-Ochiai	254.19	243.63	1901.77
	MSeer	174.57	111.40	948.47
4-fault	P-Ochiai	499.01	294.00	2300.02
	MSeer	247.03	141.50	1489.07
5-fault	P-Ochiai	570.35	322.15	2609.10
	MSeer	318.27	195.80	1644.77

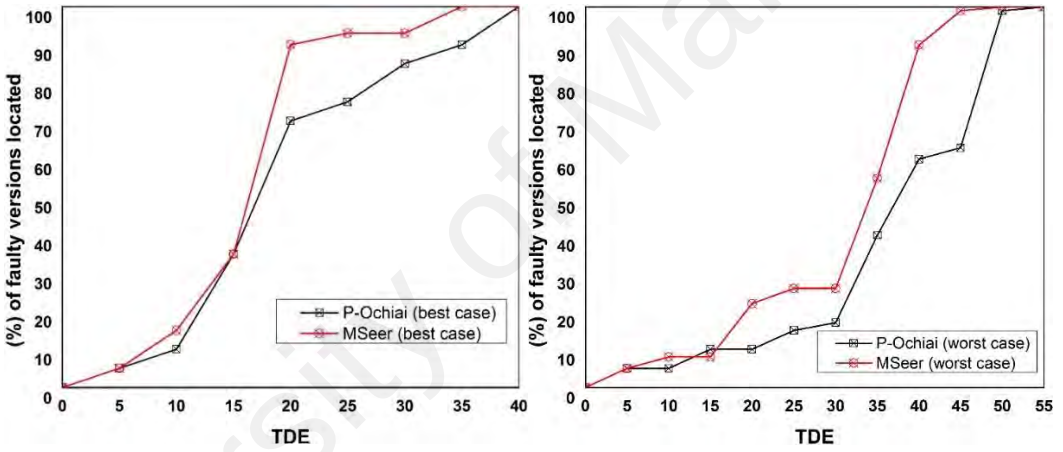
(i) ***Total developer expense (TDE)***

However, without arriving at any firm conclusion, the evaluation of MSeer and P-Ochiai with respect to the TDE score on 3-fault versions of *gzip* and *grep* subject programs is presented. In Figure 5.4 part (a) and part (b), by examining less than 10% of the program code, MSeer can locate 92% of the faulty versions in the best case and 42% in the worst case. In contrast, the TDE score for P-Ochiai when the same amount of program code is examined is 80% in the best case and 35% in the worst case. Additionally, for the best and worst case of MSeer in part (c) and part (d), by examining 10% of the code, the TDE score is 15% and 8% respectively, whereas for P-Ochiai is 10% (best case) and 5% (worst case) respectively. MSeer is consistently more effective than P-Ochiai in terms of localization effectiveness. Therefore, one can say that MSeer is far better and more effective than P-Ochiai.



(a) best case of gzip 3-fault versions

(b) worst case of gzip 3-fault versions



(c) best case of grep 3-fault versions

(d) worst case of grep 3-fault versions

Figure 5.4: TDE score-based comparison between P-Ochiai and MSeer on gzip and grep (3-fault versions).

Looking at the results in Table 5.7, Table 5.8, and Figure 5.4, one can see that MSeer is consistently more effective than P-Ochiai in terms of localization effectiveness. However, to statistically substantiate and prove MSeer superiority against P-Ochiai (the claimed problematic parallel debugging approach), Table 5.9 gives the effectiveness comparison for the best and worst cases using the Wilcoxon signed-rank test.

Table 5.9: The confidence with which it can be claimed that MSeer is more effective than P-Ochiai (best & worst cases)

		gzip		flex		grep	
		best	worst	best	worst	best	worst
2-fault	P-Ochiai	96.33%	99.15%	95.81%	98.85%	95.88%	99.85%
3-fault	P-Ochiai	97.43%	98.75%	98.85%	99.25%	94.80%	99.90%
4-fault	P-Ochiai	98.90%	99.61%	99.19%	99.35%	97.60%	99.88%
5-fault	P-Ochiai	99.45%	99.61%	99.21%	99.21%	95.57%	99.90%

From Table 5.9, one can say with 99.19% and 99.35% confidence that MSeer is more effective than P-Ochiai in best and worst cases for the 4-fault versions of *flex*. For *gzip* and *flex* programs, the confidence to accept the alternative hypothesis is higher than 95%. Likewise, for *grep* program, the confidence to accept the alternative hypothesis is at least 94% and higher in most cases. Overall, the result shows that MSeer is more effective than Ochiai which is consistent with the evaluation using the average number of statements examined and TDE score.

5.1.4. Result Summary

In this section, a brief overview of the findings of the experiments is presented.

- Firstly, based on the experiments in Section 5.1.1, it was observed that the claimed problematic parallel debugging approach is reasonably effective in locating multiple faults. However, out of the claimed problematic approach that use distinct similarity coefficient-based fault localization techniques (P-Ochiai, P-Naish2, and P-Jaccard), P-Ochiai is convincingly the most effective in locating multiple faults effectively. For instance, the average number of statements

examined by P-Ochiai, P-Naish2, and P-Jaccard for *tcas* 3-fault versions in the best case are 22.07, 29.49, and 28.02, respectively. This shows that P-Ochiai is more effective because fewer statements were examined to locate the faulty versions (25 faulty versions). Additionally, even though P-Ochiai is the most effective, the margin between the three approaches in terms of effectiveness is not as much. In conclusion, one can say that the claimed problematic parallel debugging approach is an effective multiple-fault debugging approach that can aid in reducing the total effort and time a developer requires to identify the locations of faults.

- Secondly, in Section 5.1.2, it was observed that P-Ochiai is more effective than OBA debugging approach. For example, on *sed* 2-fault versions, by examining less than 10% of the program code, P-Ochiai can locate 75% and 35% of the faulty versions in both the best and worst cases, respectively. In contrast, OBA can locate 70% and 30% of all the faulty versions in the best and worst cases, respectively. Furthermore, it was observed that by utilizing the claimed problematic parallel debugging approach (P-Ochiai), *fault-focused* clusters are often generated redundantly that target the same fault which reduces the approach's effectiveness greatly. Generally, one can confidently conclude that based on the experimental results, P-Ochiai is relatively more effective than OBA debugging approach irrespective of the clustering algorithm and distance metric used.
- Lastly, MSeer outperforms P-Ochiai in all the subject programs compared with in both the best and worst case scenarios as shown in Section 5.1.3. Looking at Figure 5.4, *gzip* program's faulty versions, by examining less than 10% of the program code, MSeer locates 92% of the faulty versions in the best case and 42% in the worst case. In contrast, the TDE score for P-Ochiai when the same amount of program code is examined is 80% in the best case and 35% in the worst case.

Therefore, regardless of whether the best or worst case is considered, MSeer is consistently the most effective in all the faulty versions. Although P-Ochiai is more effective than OBA, yet in comparison with MSeer, it is not that effective. Furthermore, it was observed that estimating the number of clusters based on the number of failed test cases as highlighted in Chapter 3, Section 3.2.2 is indeed not appropriate as later recognized by Gao et al. (R. Gao & Wong, 2017) because there is no clear correlation between the number of failed test cases and the number of faults in a given program. Therefore, many redundant clusters can be generated that do not target faults which will increase the time and effort for a developer to look for faults. In conclusion, clustering failed test based on their execution profile similarity and the utilization of distance metrics such as Euclidean distance to measure the *due-to* relationship between failed tests is indeed problematic and contributes to the reduction of effectiveness of a parallel debugging approach.

5.2. Experiment 2

This section discusses the experimental results of the proposed technique named multiple fault localization based on complex network theory (FLCN-M) presented in Chapter 4, Section 4.2.2. The experiment in this section is evaluated based on two evaluation metrics, namely EXAM Score and IDE score.

5.2.1. Effectiveness of FLCN-M on Single-Fault Programs

Figure 5.5 depicts the percentage of located faults in terms of EXAM Score. Apart from evaluating the proposed FLCN-M technique on multiple-fault subjects, the following fault localization techniques which are generally known as some of the best techniques on single-fault programs, namely Tarantula (J. A. Jones & Harrold, 2005), Sober (Liblit et al., 2005), Delta Debugging (DD) (Zeller, 2002), Nearest Neighbour (NN) (Renieres & Reiss, 2003), SNCM (Zhu et al., 2011), Intersection, and Union (Renieres & Reiss, 2003), are also plotted for the comparative analysis for the evaluation on single-fault programs. For SNCM, it was run in the same environment of the proposed technique. The values of the other techniques are directly cited from their respective papers.

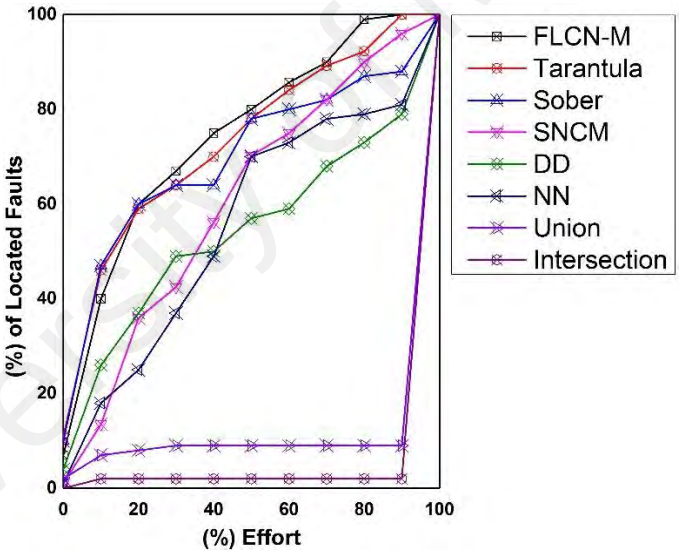


Figure 5.5: Effectiveness comparison between FLCN-M and other fault localization techniques on single-fault programs (Siemens test suite)

Based on the results shown in Figure 5.5, it was observed that FLCN-M performs relatively better with a very slight margin, followed by Tarantula and Sober. FLCN-M is capable of finding 90% of the faulty versions by examining less than 70% of the programs' code and also finding about 99% of the faulty versions by examining less than

80% of the programs' code. SNCM technique which uses centrality measures to locate faults is clearly outperformed by the proposed technique.

However, the performance of the proposed FLCN-M technique is less ideal in the initial stage, where by checking less than 10% of the programs' code, a developer can only locate 40% of faults in all faulty versions. Tarantula and Sober, on the other hand, are capable of locating 46% and 47% of faults with the same effort, respectively. It was observed that this is mainly due to the sensitivity of the proposed FLCN-M technique with statements executed by passed test cases. Therefore, the effectiveness will probably improve if the program network is modeled with failed test cases only instead of considering both tests executions.

5.2.2. Effectiveness of FLCN-M on SIEMENS-M

Figure 5.6 depicts the results for 2-fault versions of SIEMENS-M. The results show that FLCN-M can locate 10% of the faults by examining less than 10% of the program code. The IDE score of the 2-fault versions is 70%, meaning by checking less than 70% of all the faulty versions containing 2 faults, all the faulty versions can be found by the developer. However, Ochiai can at best locate all the faulty versions with 2 faults by checking less than 80% of the code. With 3 faults, as shown in Figure 5.7, the developer can find 50% of the faults by examining less than 10% of the faulty versions using the FLCN-M technique, while by utilizing Ochiai, the developer can locate 30% of the faults by examining the same amount of faulty versions.

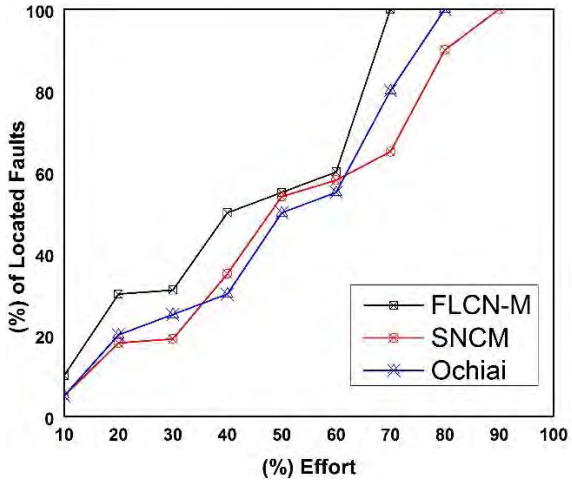


Figure 5.6: IDE score-based comparison on 2-fault versions

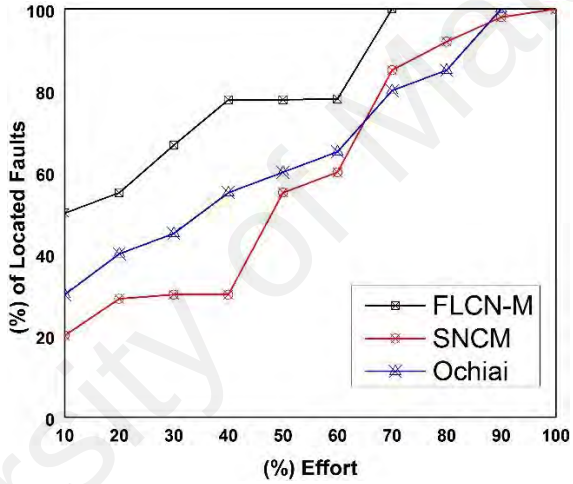


Figure 5.7: IDE score-based comparison on 3-fault versions

Furthermore, using the proposed technique (FLCN-M), Figure 5.8 (4-fault) shows that 50% of faults can be found by examining less than 20% of the faulty versions. The IDE score reduces even further in Figure 5.9 with 5 faults where by examining less than 45% of program code, 100% of the faults can be found in all the faulty versions. In contrast, SNCM achieves higher expense in all the multiple-fault experiments, while Ochiai performs relatively better in comparison with SNCM technique.

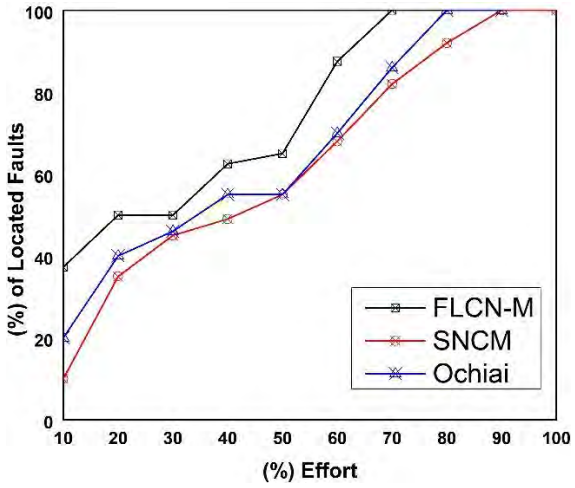


Figure 5.8: IDE score-based comparison on 4-fault versions

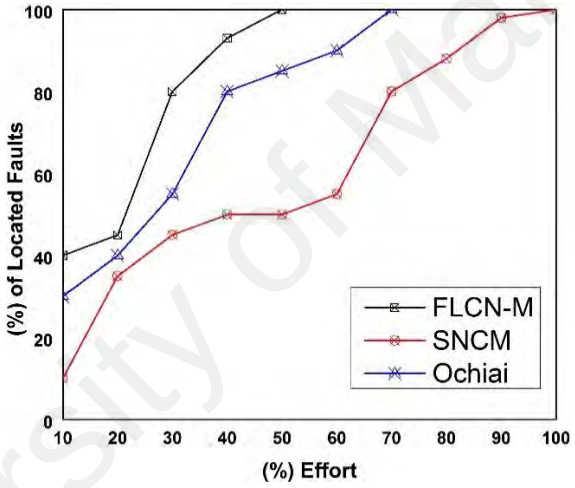


Figure 5.9: IDE score-based comparison on 5-fault versions

The results show a trend where the more faults exist in a program, the more effective the proposed technique is in locating faults. Moreover, even if passed tests execution execute faulty statements in a high proportion (DiGiuseppe & Jones, 2011b; Xue & Namin, 2013), the proposed technique is still capable of localizing those faulty program statements simultaneously with relatively good effectiveness compared to the baseline techniques. As a result, developer expense can be reduced even if the number of faults increases using the proposed technique (FLCN-M) because it takes into account both passed and failed tests executions of a given faulty program.

5.2.3. Effectiveness of FLCN-M on UNIX Real-life Utility Programs

The Unix real-life utility programs (*gzip* and *sed*) are composed of both real and seeded faults in each faulty version. To evaluate the effectiveness of the proposed technique, it is compared with Tarantula, Ochiai, and SNCM fault localization techniques. As discussed earlier, all faults are to be localized simultaneously in a single diagnosis rank list. In Figure 5.10, FLCN-M locates 45% of the faulty versions by examining less than 10% of program code and by examining less than 40% of the program code, a developer can locate 95% of all faults in the faulty versions in a single diagnosis rank list. However, Tarantula and Ochiai can locate 30% and 35% of the faulty versions by examing less than 10% of the program code, respectively.

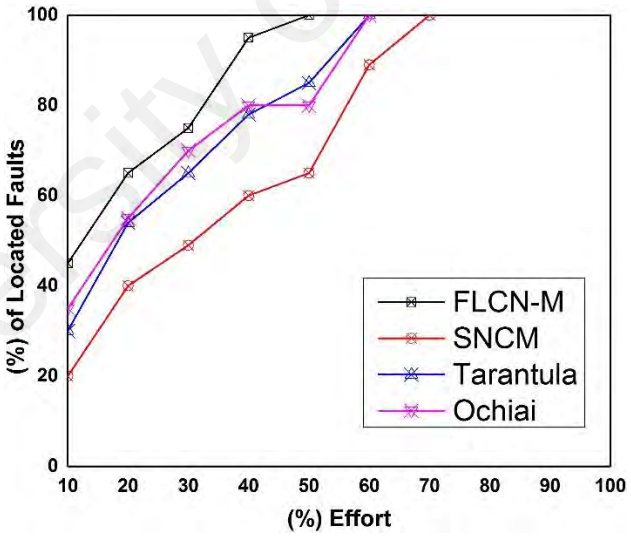


Figure 5.10: IDE score-based comparison between FLCN-M, Tarantula, Ochiai, and SNCM on gzip program

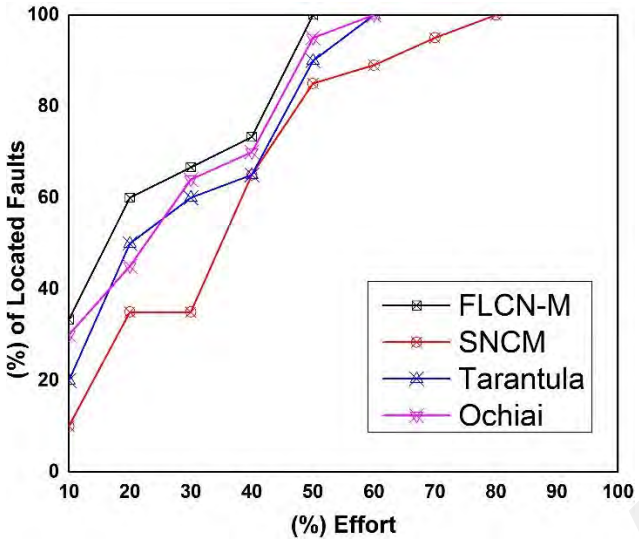


Figure 5.11: IDE score-based comparison between FLCN-M, Tarantula, Ochiai, and SNCM on sed Program

As shown in Figure 5.11, FLCN-M outperforms Tarantula, Ochiai, and SNCM where it is capable of identifying all the locations of faulty statements by examining less than 50% of the faulty versions. Using a simultaneous approach to debugging (locating all the faults in a single diagnosis rank list) caused the other techniques to lose their effectiveness. In general, the proposed FLCN-M technique surpasses Tarantula, Ochiai, and SNCM in locating faults simultaneously in a single diagnosis rank list.

5.2.4. Impact of Centrality Measures on the Proposed FLCN-M Technique

With regards to the assertion that program statements with higher degree centrality are related to a fault. Figure 5.12 shows the percentage of faulty statements found with respect to their degree centrality (*Dc*) value in each faulty version of the SIEMENS-M programs with *Dc* of 3 and *Dc* of 2. Figure 5.12 illustrates that in SIEMENS-M programs with 2-fault, 20% of the faults were located in program statements with *Dc* of 3 while

80% of faults are found in statements with *Dc* of 2. Statements with *Dc* lower than 2 have no faults across all faulty versions of the experiments.

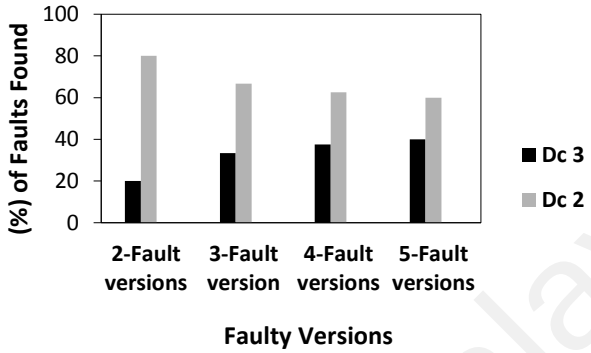


Figure 5.12: SIEMENS-M (Degree centrality of program statement and its correlation with failure)

In 3-fault versions, 33.33% of faults are found in statements with *Dc* of 3, while 66.67% are found in statements with *Dc* of 2. In 4-fault versions, 37.5% of faults are located in program statements with *Dc* of 3 and 62.5% in statements with *Dc* of 2; while in 5-fault versions, 40% of faults are found in statements with *Dc* of 3 and 60% in statements with *Dc* of 2. However, it was observed that the higher the number of faults in a given multiple-fault program, the number of faulty statements with high *Dc* value increases. This implies that faulty statements will be ranked at the top of the diagnosis rank list, and in return, help in simultaneous localization of multiple faults.

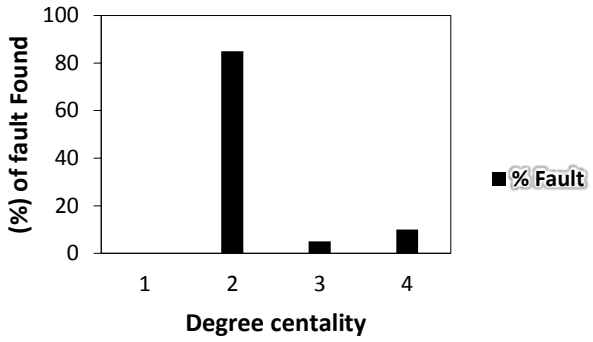


Figure 5.13: Gzip (Degree centrality of program statement and its correlation with failure)

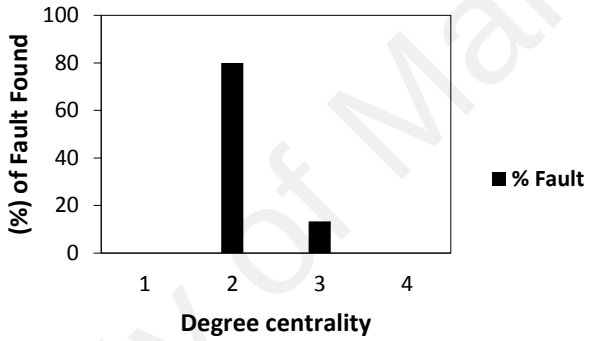


Figure 5.14: Sed (Degree centrality of program statement and its correlation with failure)

Figure 5.13 and Figure 5.14 show *Dc* correlation with failure across all faulty versions of both *gzip* and *sed* programs, respectively. For *gzip* (Figure 5.13), program statements with *Dc* values of 4, 3, and 2 contain 10%, 5%, and 85% of the faults in all faulty versions respectively, while for *sed* program (Figure 5.14), 80% of the faults are found in program statements with *Dc* of 2, while 20% of the faults can be found in program statements with *Dc* of 3. The trend shows that on average, 60% - 70% of all the faults are located in program statements with *Dc* of 2.

5.3. Experiment 3

This section presents the experimental results of the proposed technique named single fault localization based on complex network theory (FLCN-S) presented in Chapter 4, Section 4.2.3. The results of the comparison with the baseline similarity coefficient-based fault localization techniques, namely Ochiai coefficient and Jaccard coefficient are also highlighted and discussed. The experiment in this section is evaluated based on three evaluation metrics, namely cumulative number of statements examined, EXAM Score, and Wilcoxon signed-rank test. For all the subject programs used in this experiment, each faulty version has exactly one fault. For the experiment in this section, it is presumed that the best case effectiveness entails that the faulty statement is identified at the top of the list of statements having the same suspiciousness score values, while for the worst case effectiveness, the faulty statement resides at the bottom of the list having the same suspiciousness score values. In the evaluation of FLCN-S technique, the result is mostly presented between these two levels of effectiveness for all the evaluation metrics (except for the result presented in Figure 5.16).

5.3.1. Effectiveness of FLCN-S on Siemens Test Suite Programs

Table 5.10 presents the cumulative (total) number of statements examined by FLCN-S and the baseline techniques in both the best and worst cases. For each program, in the best case scenarios, FLCN-S requires the examination of fewer statements than the compared techniques. The same applies to the worst case scenarios. For instance, it was observed that for *print_tokens* program, FLCN-S can locate all the faulty versions by examining no more than 251 statements in the best case scenario, and 699 in the worst case scenario, respectively. Furthermore, with respect to the *print_tokens* program, the

second best technique is Ochiai, which requires the examination of no more than 324 statements in the best case and 712 statements in the worst case scenario.

It is worth knowing that these values represent the total number of statements that each technique requires to examine to locate the faults in each subject program. Looking at Table 5.10, it was observed that irrespective of which scenario is considered (best case or worst case), FLCN-S is consistently the most effective technique. Another important point worth noting is that, Ochiai is more effective than Jaccard in all cases (best and worst cases). Henceforth, it is worth re-emphasizing that FLCN-S is the most effective technique with respect to both the best and worst case scenarios.

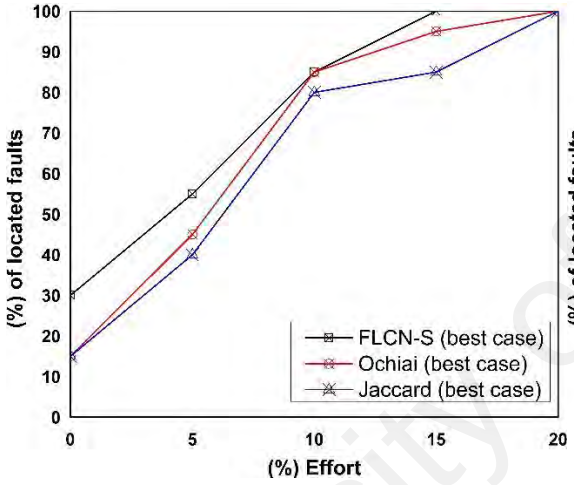
Table 5.10: Cumulative number of statements examined to locate faults for each program in Siemens test suite (best & worst cases)

	FLCN-S		Ochiai		Jaccard	
	Best case	Worst case	Best case	Worst case	Best case	Worst case
tcas	195	388	205	408	259	500
print_tokens	251	699	324	712	404	799
print_tokens2	408	600	423	653	451	708
schedule	311	655	363	702	388	750
schedule2	499	583	550	627	561	641
replace	340	455	370	500	401	513
tot_info	200	350	270	420	304	608

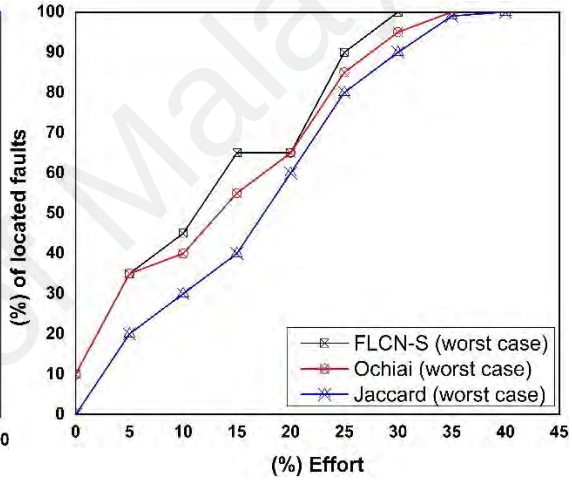
However, without arriving at any firm conclusion, the evaluation of FLCN-S with respect to EXAM Score is presented. The single-fault versions of *tcas, print_tokens,* and *print_tokens2* in the best and worst cases are highlighted in Figure 5.15. The figure shows the effectiveness of FLCN-S in comparison with two other techniques, namely Ochiai and Jaccard. The *y*-axis indicates the percentage of faults located in all the program's

faulty versions, while the *x*-axis indicates the effort spent to locate the corresponding faults.

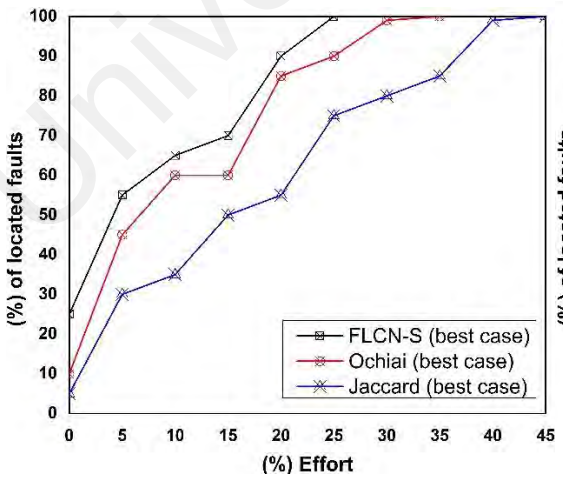
For instance, based on part (a) and (b) of Figure 5.15, it was observed that on the *tcas* program, by examining less than 10% of the program code, FLCN-S can locate 85% of the faults in the best case, and 45% in the worst case. Correspondingly, by examining the same amount of code (less than 10%), Ochiai (the second best) can only locate 85% (best case) and 40% (worst case).



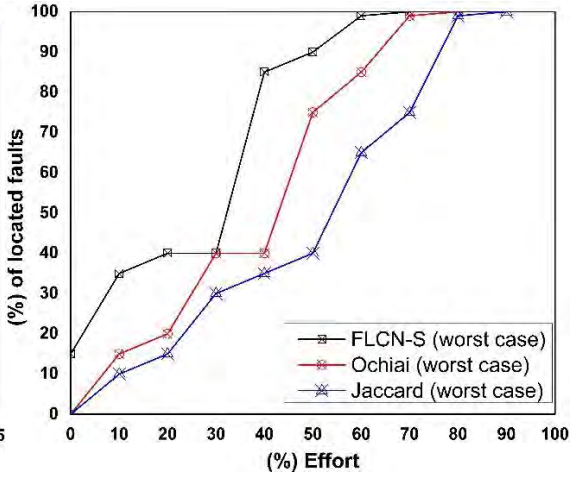
(a) best case of tcas



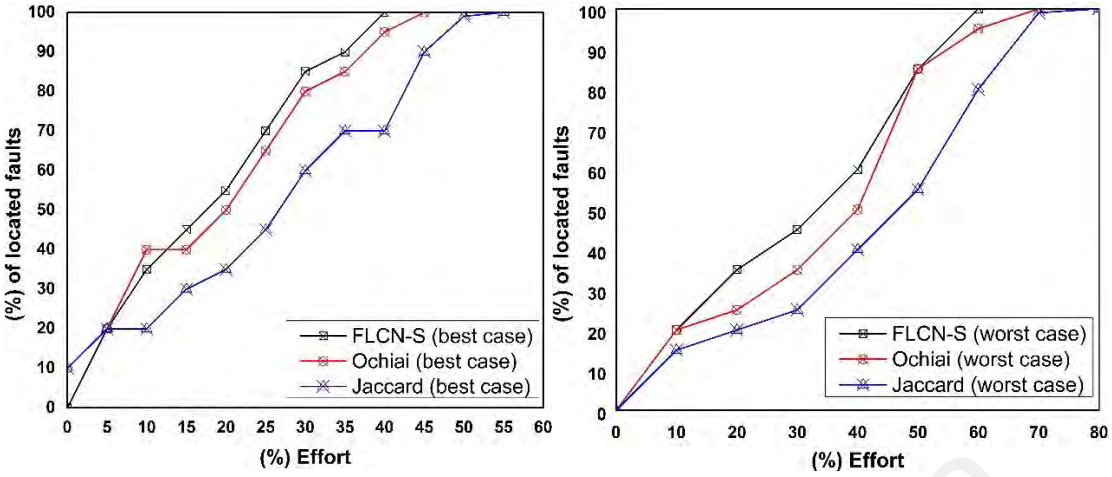
(b) worst case of tcas



(c) best case of print_tokens



(d) worst case of print_tokens



(e) best case of print_tokens2 (f) worst case of print_tokens2

Figure 5.15: EXAM score-based comparison between FLCN-S and the baseline techniques on tcas, print_tokens, and print_tokens2.

In part (c) and (d), the effectiveness score of *print_tokens* is presented. It was observed that, by examining less than 10% of the program code, FLCN-S can locate 65% of the faulty versions in the best case and 35% in the worst case. Ochiai (the second best) can locate 60% of the faults in the best case, and 15% in the worst case. Moreover, the percentage for Jaccard (the third best) is 35% (best case), and 10% (worst case).

Furthermore, in part (e) and (f) of Figure 5.15, with respect to EXAM Score, FLCN-S performs relatively better. The curves show that by examining less than 20% of the program code, FLCN-S can locate 55% of the faulty versions in the best case and 35% in the worst case. Ochiai (the second best) can only locate 50% in the best case and 25% in the worst case when examining the same amount of code. For Jaccard (the third best), by examining the same amount of code (less than 20%), it is 35% (best case), and 20% (worst case).

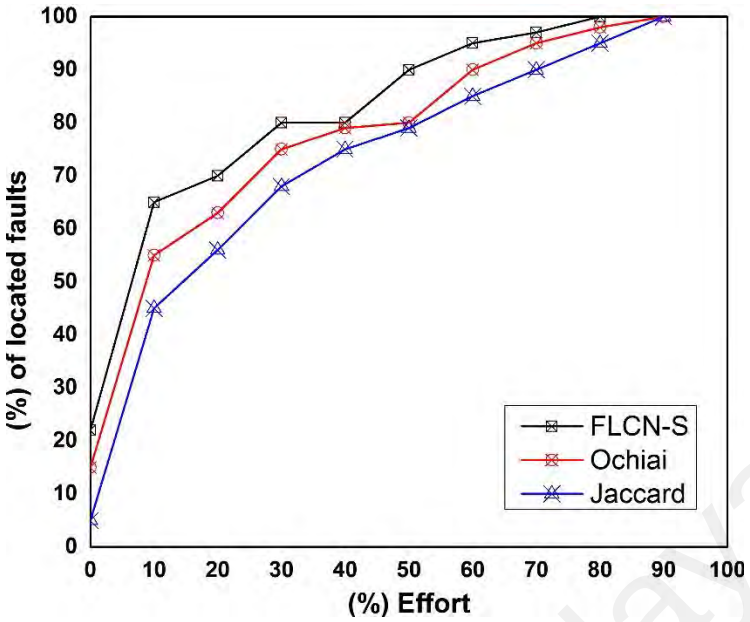


Figure 5.16: Overall effectiveness comparison on Siemens test suite

Figure 5.16 gives the overall effectiveness score of FLCN-S with the baseline fault localization techniques on all Siemens test suite programs. Based on the result, FLCN-S can locate 65% of the faults in all faulty versions of the Siemens test suite by examining less than 10% of the programs' code. It was observed that FLCN-S has yielded a drastic improvement from the benchmark technique (FLCN-M) (Zakari, Lee, & Chong, 2018) with a 25% increase in located faulty versions by checking less than 10% of the programs' code. Furthermore, FLCN-S has also outperformed Zoltar-S approach as concluded by Abreu et al. (Abreu et al., 2011), where the latter can only locate 60% of the faults by checking less than 10% of the programs' code.

Table 5.11: The confidence with which it can be claimed that FLCN-S is more effective than Ochiai and Jaccard on Siemens test suite programs (best & worst cases)

	Ochiai		Jaccard	
	Best case	Worst case	Best case	Worst case
tcas	90.00%	95.00%	98.44%	99.11%
print_tokens	98.64%	92.31%	99.35%	99.00%

Table 5.11, continued

print_tokens2	93.33%	98.08%	97.68%	99.08%
schedule	97.88%	97.88%	98.71%	98.95%
schedule2	98.04%	97.73%	98.39%	98.28%
replace	96.67%	97.78%	98.37%	98.28%
tot_info	98.58%	98.58%	99.04%	99.62%

Based on the third evaluation metric, Table 5.11 gives the effectiveness comparisons of FLCN-S with Ochiai and Jaccard using the Wilcoxon signed-rank test. The entries in the table give the confidence of which the alternative hypothesis (which implies that FLCN-S requires the examination of fewer statements than the compared baseline techniques to locate faults) can be accepted. For example, one can say with 97.88% confidence that FLCN-S is more effective than Ochiai on *schedule* program in both best and worst cases. Nevertheless, for *schedule2, replace,* and *tot_info* programs, the confidence to accept the alternative hypothesis is higher than 96% in all scenarios (best & worst cases).

Few scenarios have the confidence level that is lesser than 95%. For instance, FLCN-S being more effective than Ochiai with 90.00% confidence for the best case of *tcas*, 93.33% confidence being better than Ochiai for the best case of *print_tokens2*, and with 92.31% confidence being better than Ochiai for the worst case of *print_tokens*. In summary, the results from the Wilcoxon signed-rank test clearly shows that FLCN-S is more effective than the compared baseline techniques on Siemens test suite subject programs. The result is also in line with the former conclusion that FLCN-S performs better than the compared techniques in terms of the cumulative number of statements examined and the EXAM Score.

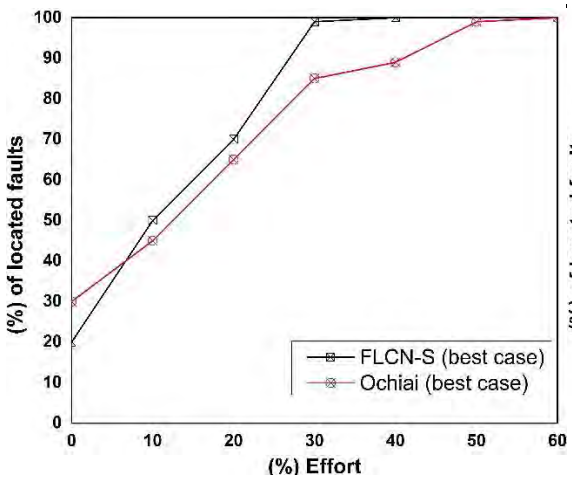
5.3.2. Effectiveness of FLCN-S on UNIX Real-life Utility Programs

Table 5.12 gives the total number of statements examined by FLCN-S and Ochiai across two programs (*gzip* and *sed*) to locate all faults in the programs' faulty versions. Each faulty version under consideration has exactly one fault for this experiment. From Table 5.12, it was observed that in all scenarios (best and worst cases), FLCN-S is always the most effective in comparison with Ochiai. For example, the total number of statements examined by FLCN-S on *sed* program is 3201 in the best case, and 4100 in the worst case. On the other hand, Ochiai is 3885 in the best case, and 4652 in the worst case.

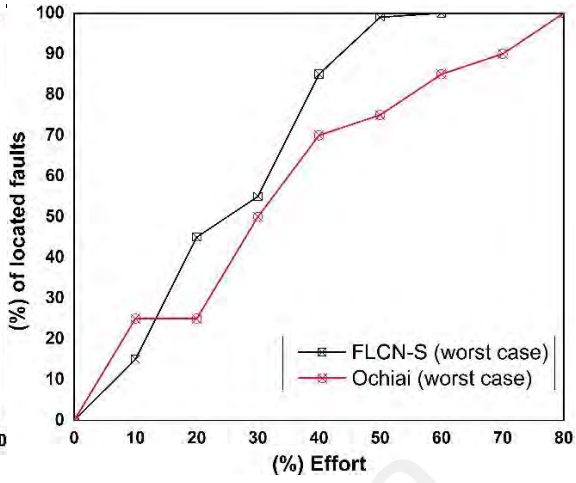
Table 5.12: Cumulative number of statements examined by FLCN-S and Ochiai (best & worst cases)

	Best Case		Worst Case	
	FLCN-S	Ochiai	FLCN-S	Ochiai
gzip	1944	2692	2770	3992
sed	3201	3885	4100	4652

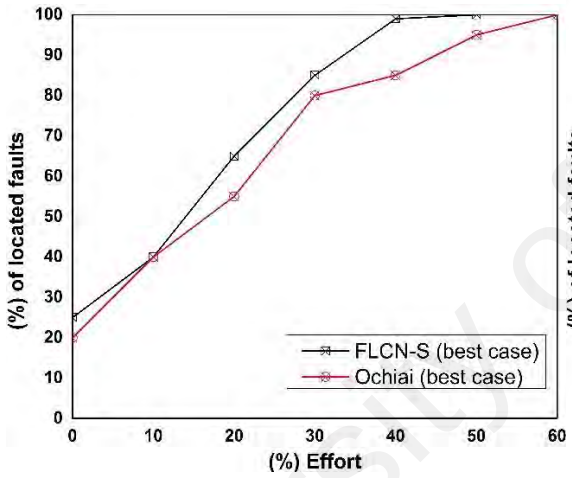
Next, the evaluation of FLCN-S with respect to EXAM Score is given. In Figure 5.17, the best and worst cases of *gzip* and *sed* programs are presented. The figure shows the effectiveness on FLCN-S in comparison with Ochiai. The black curve represents FLCN-S while the red curve represents Ochiai. Looking at part (a) and (b) of Figure 5.17, one can find that on *gzip* program, by examining less than 30% of the program code, FLCN-S can locate 99% of the faulty versions in the best case and 55% in the worst case. In contrast, by examining the same amount of program code, Ochiai can only locate 85% of faults in the best case and 50% in the worst case.



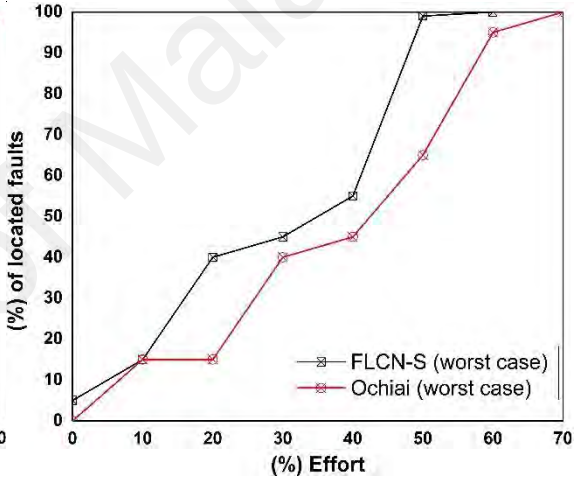
(a) best case of gzip



(b) worst case of gzip



(c) best case of sed



(d) worst case of sed

Figure 5.17: EXAM score-based comparison between FLCN-S and Ochiai on gzip, and sed.

In part (c) and (d), the effectiveness score of *sed* program is presented. The curves show that by examining less than 20% of the program code, FLCN-S can locate 65% of faults in the best case and 40% in the worst case, respectively. Correspondingly, by examining the same amount of program code, Ochiai can locate 55% (best case) and 15% (worst case). The conclusion drawn from Figure 5.17 with respect to EXAM Score of both techniques (FLCN-S and Ochiai) is that FLCN-S performs better than Ochiai. This

result is consistent with the observations from Table 5.12 that FLCN-S is the most effective technique. Looking at the third evaluation metric, Table 5.13 highlights data comparing FLCN-S with Ochiai using Wilcoxon signed-rank test. The table highlights the confidence to which the alternative hypothesis can be accepted. For instance, one can say with 99.86% (best case) and 99.82% (worst case) confidence that FLCN-S is more effective than Ochiai on the *sed* program.

Table 5.13: The confidence with which it can be claimed that FLCN-S is more effective than Ochiai (best & worst cases)

	Ochiai (Best)	Ochiai (Worst)
gzip	99.87%	99.92%
sed	99.86%	99.82%

Generally, for *gzip* and *sed* programs, the confidence to accept the alternative hypothesis is higher than 99%. In total, the results from this test (Wilcoxon signed-rank test) show that FLCN-S is more effective than Ochiai which is consistent with the former results using EXAM Score and the cumulative (total) number of statements examined.

5.3.3. Impact of Centrality Measures on the Proposed FLCN-S Technique

In the earlier experiment in Section 5.2.4, the study on the impact of degree centrality and how statement degree relates to faults was conducted. It was concluded that statement degree is vital in identifying faulty program statements, especially in the multiple-fault context. In this section, it was observed that on Siemens test suite programs (single fault), 23% of all faulty statements have *Dc* of 3, while 77% have *Dc* of 2. Moreover, it was observed that in both single-fault and multiple-fault contexts, closeness centrality plays a

vital role in the ranking and identification of faulty program statements. Moreover, on both Siemens test suite and UNIX real-life utility programs, both degree centrality and closeness centrality play a critical and vital role in the identification of faulty program statements.

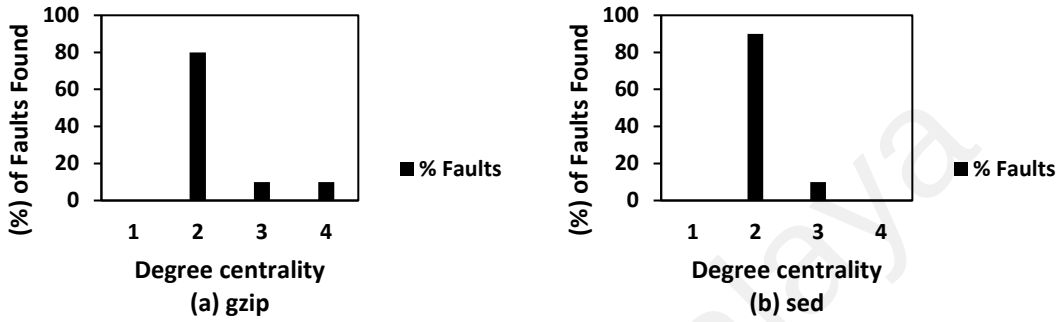


Figure 5.18: Degree centrality correlation with failure for UNIX real-Life utility programs.

In Figure 5.18, there was a smaller number of faults that were located on program statements with *Dc* of 4. Hence, almost 80% of all the faults are located on statements with *Dc* of 2 for both programs (*gzip* and *sed*). In Figure 5.19, it was observed that most of the faults are located on the program statement with *Dc* of 2. However, a significant small number of faults were located on programs with *Dc* of 3 and 4. Degree centrality is a single factor when localizing faults using the proposed technique (FLCN-S) with closeness centrality playing a critical role in the fault localization process. This analysis is aimed at confirming the claims of previous studies in various research domains where researchers indicated the important role degree centrality plays in the identification of the most influential and faulty nodes in a network (Borgatti, 2005; Cai & Yin, 2009; Girvan & Newman, 2002; Li, Han, & Hu, 2008; Zhu et al., 2011). Hence, it was observed that both degree centrality and closeness centrality play a vital role in the identification of faulty program statements.

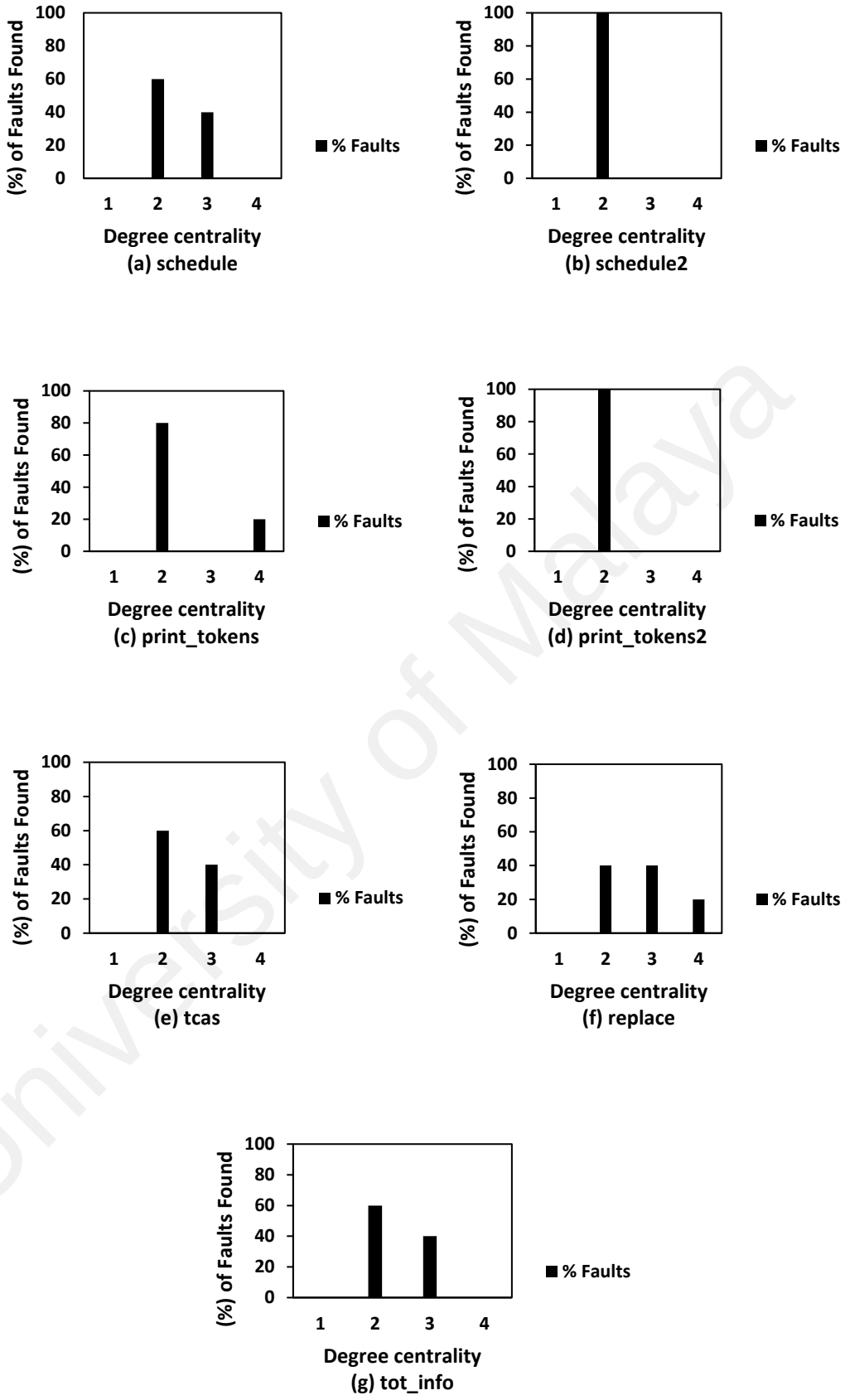


Figure 5.19: Degree centrality correlation with failure for Siemens suite programs.

5.3.4. Overall Observations

Generally, the effectiveness of a given technique is not always constant and can change depending on the subject program used. It was observed that by utilizing failed test inputs alone, the effectiveness of the proposed fault localization technique (FLCN-S) has increased on single-fault programs. In the initial work (FLCN-M), both test cases (passed/failed) were utilized to localize single faults, the accuracy was not convincing, where 40% EXAM Score was achieved by checking less than 10% of the program faulty versions on Siemens test suite programs (single-fault programs). Therefore, it was concluded that by utilizing failed test input alone, the accuracy of the proposed technique (FLCN-S) increases in the context of single fault due to the minimal fault-to-failure complexity that affects localization on multiple-fault programs. The technique can effectively localize 65% of all faulty versions on Siemens test suite subjects by checking less than 10% of the program code and is largely more effective on Unix real-life utility program in comparison with the baseline techniques in both best and worst case scenarios. Finally, it was also observed that both degree centrality and closeness centrality play a vital role in the identification of faulty program statements.

5.4. Experiment 4

This section presents the experimental results of the new community-based fault isolation approach that aids in the effective isolation and localization of multiple faults simultaneously in parallel as presented in Chapter 4, Section 4.2.4. The experiment in this section is evaluated based on three metrics, namely the average number of statements examined, TDE score, and Wilcoxon signed-rank test. For all the experiments, it is presumed that for the best case effectiveness, the faulty statement is at the very top of the

suspicious statements ranking list; and for the worst case effectiveness, the faulty statement is at the very end of the ranking list.

5.4.1. Cross-Comparison with P-Ochiai

This section provides the results for the cross-comparison between the proposed approach and P-Ochiai (the claimed problematic parallel debugging approach).

(j) Average number of statements examined

Table 5.14 and Table 5.15 present the average number of statements examined by both the proposed approach and P-Ochiai with respect to the best and worst cases. The average number of statements examined by the approaches are based on 25 versions of a given program each containing x number of faults (x = 2, 3, 4, and 5). It was observed that the average number of statements examined by the proposed approach on 2-fault faulty versions of *flex* is 9.08 in the best case, and 62.18 in the worst case. On the other hand, for P-Ochiai, the best case is 35.23, and the worst case is 161.08. For the 3-fault faulty versions of *tcas*, the average number of statements examined by the proposed approach is 11.04 in the best case and 22.33 in the worst case. With respect to P-Ochiai, the average number of statements examined in the same faulty versions (*tcas*, 3-fault versions) to locate the faults is 22.07 in the best case, and 28.49 in the worst case. From both tables (Table 5.14 and Table 5.15), it was observed that, regardless of whether the best case or worst case is considered, the proposed approach is always the most effective in comparison with P-Ochiai. The proposed approach has shown to be generally more effective than P-Ochiai, however, it is not surprising due to P-Ochiai's obvious limitations

where the approach clusters failed tests execution based on their execution profile similarity. With the tests representation and the distance metric that the approach utilized (P-Ochiai), less effective localization inferencing is expected. But it is worth re-emphasising that the proposed approach is by far the most effective in comparison with P-Ochiai.

Table 5.14: Average number of statements examined (best case)

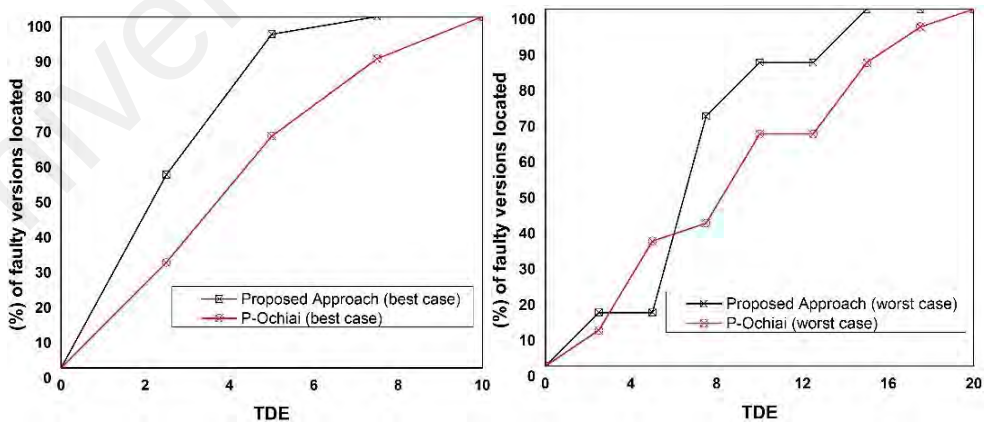
		tcas	replace	gzip	sed	flex	grep
2-fault	Proposed Approach	5.97	12.03	10.87	50.21	9.08	333.99
	P-Ochiai	10.09	19.14	40.03	81.48	35.23	370.01
3-fault	Proposed Approach	11.04	20.18	25.66	101.18	30.18	450.39
	P-Ochiai	22.07	24.05	80.63	393.89	110.03	512.15
4-fault	Proposed Approach	18.11	39.06	60.00	280.38	60.07	525.33
	P-Ochiai	33.33	60.04	161.04	500.10	190.00	601.60
5-fault	Proposed Approach	28.14	85.11	77.18	549.35	95.83	549.35
	P-Ochiai	49.04	98.91	260.03	679.74	230.06	621.34

Table 5.15: Average number of statements examined (worst case)

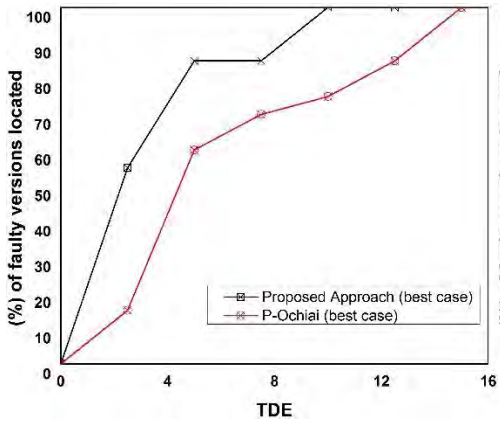
		tcas	replace	gzip	sed	flex	grep
2-fault	Proposed Approach	13.03	27.89	80.89	200.13	62.18	621.76
	P-Ochiai	17.49	46.41	194.33	292.09	161.08	1310.01
3-fault	Proposed Approach	22.33	38.03	150.14	450.03	101.04	888.89
	P-Ochiai	28.49	55.19	254.19	601.75	243.63	1901.77
4-fault	Proposed Approach	40.99	75.99	231.05	701.14	150.23	1340.70
	P-Ochiai	42.89	100.89	499.01	889.05	294.00	2300.03
5-fault	Proposed Approach	46.11	120.83	310.11	1125.55	186.99	1599.14
	P-Ochiai	56.02	155.97	570.35	1501.08	322.15	2609.10

(k) Total developer expense (TDE)

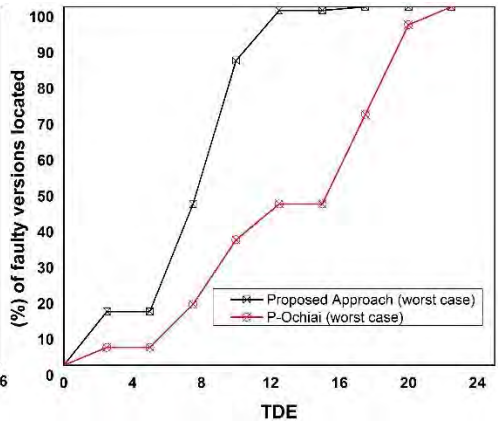
Now the evaluation of the proposed approach with respect to TDE score is presented for all faulty versions. In Figure 5.20, the 2-fault versions of *gzip, sed,* and *flex* in best and worst cases are presented. The *y*-axis represents the percentage (%) of faulty versions located by a developer while the *x*-axis represents the percentage of code examined (total developer expense). Looking at part (a) and (b) of Figure 5.20, it was found that on the *gzip* program, by examining less than 7.5% of the program code, the proposed approach can locate all the faulty versions in the best case, and 70% in the worst case. In contrast, by examining the same amount of code, P-Ochiai can only locate 88% of faults in the best case, and 40% in the worst case. In part (c) and (d), the effectiveness score of *sed* 2-fault faulty versions are presented. The curves show that by examining less than 10% of the program code, the developer can locate all the faulty versions in the best case with the proposed approach, while P-Ochiai can locate 75% of the faulty versions. In the worst case, by examining the same amount of code (less than 10%), 85% of the faulty versions can be located using the proposed approach, and 35% with P-Ochiai.



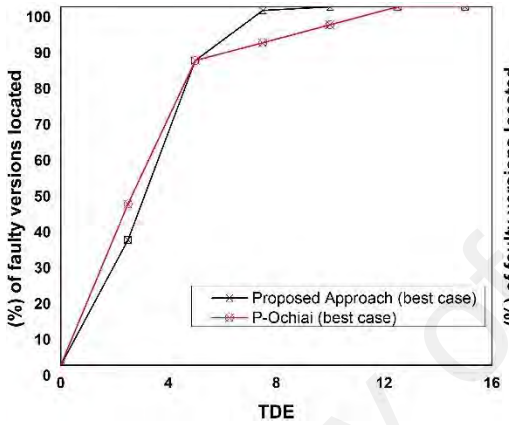
(a) best case of gzip 2-fault versions (b) worst case of gzip 2-fault versions



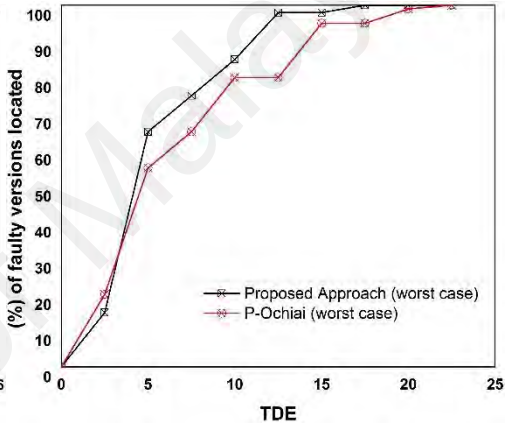
(c) best case of sed 2-fault versions



(d) worst case of sed 2-fault versions



(e) best case of flex 2-fault versions



(f) worst case of flex 2-fault versions

Figure 5.20: TDE score-based comparison between the proposed approach and P-Ochiai on gzip, sed, and flex (2-fault versions).

Consistently, looking at part (e) and (f) of Figure 5.20 (*flex* program), the proposed approach is still the most effective. However, it is worth highlighting that in some cases, the TDE score difference between the two approaches is not much. For example, with respect to *flex* program part (e) in the best case, the TDE score difference is 2.5% where by using the proposed approach all the faulty versions can be located by examining less than 10% of the code, and using P-Ochiai, all the faulty versions can be located by examining 12.5% of the program code. Looking at Figure 5.20, the story is the same as

in Table 5.14 and Table 5.15 where the proposed approach consistently surpasses P-Ochiai in locating faults effectively.

Having looked at the results in terms of the average number of statements examined and TDE score, Table 5.16 and Table 5.17 give the effectiveness comparisons of the best and worst cases using the Wilcoxon signed-rank test. The tables give the confidence of which the alternative hypothesis can be accepted (that the proposed approach requires the examination of fewer statements than the compared baseline approach). For example, it can be said with 99.46% (best case) and 99.62% (worst case) confidence that the proposed approach is more effective than P-Ochiai on 5-fault versions of *gzip*. For *gzip, sed, flex,* and *grep* programs, the confidence to accept the alternative hypothesis is higher than 96% in both best and worst cases across all faulty versions (2-fault, 3-fault, 4-fault, and 5-fault).

Table 5.16: The confidence with which it can be claimed that the proposed approach is more effective than P-Ochiai (best cases)

		tcas	replace	gzip	sed	flex	grep
2-fault	P-Ochiai	75.73%	85.94%	96.58%	96.81%	96.18%	97.23%
3-fault	P-Ochiai	90.94%	74.17%	98.19%	99.66%	98.75%	98.39%
4-fault	P-Ochiai	93.43%	95.24%	99.02%	99.55%	99.24%	98.69%
5-fault	P-Ochiai	95.22%	92.76%	99.46%	99.26%	99.26%	98.62%

Table 5.17: The confidence with which it can be claimed that the proposed approach is more effective than P-Ochiai (worst cases)

		tcas	replace	gzip	sed	flex	grep
2-fault	P-Ochiai	77.58%	94.61%	99.12%	98.92%	98.99%	99.86%
3-fault	P-Ochiai	83.77%	94.18%	99.04%	99.35%	99.30%	99.91%
4-fault	P-Ochiai	47.37%	95.99%	99.63%	99.47%	99.31%	99.90%
5-fault	P-Ochiai	89.91%	97.16%	99.62%	99.74%	99.27%	99.91%

However, for *tcas* and *replace* programs across all the faulty versions on the best and worst cases scenarios, the alternative hypothesis is accepted with a confidence level of higher than 90% in most cases with a few exceptions having confidence level as low as 47.37% (in the worst case of *tcas* program 4-fault versions). The scenarios with low confidence level are where the difference between the proposed approach and P-Ochiai in terms of statements examined to locate all the faulty versions by each approach is very small. In totality, the result shows that the proposed approach performs better than P-Ochiai in both best and worst cases.

5.4.2. Cross-Comparison with MSeer and P-Ochiai

This section provides the results for the cross-comparison between the proposed approach, P-Ochiai, and MSeer parallel debugging approach.

(l) Average number of statements examined

With respect to 25 versions of three programs (*gzip, flex,* and *grep*) containing x number of faults ($x = 3$), Table 5.18 gives the average number of statements examined by the proposed approach, MSeer, and P-Ochiai in both best and worst cases to produce a failure-free program. For instance, the average number of statements examined by the proposed approach on *grep* is 450.39 in the best case, and 888.89 in the worst case. On the other hand, MSeer is 492.93 in the best case, 948.47 in the worst case. For P-Ochiai, the best case is 512.15, and the worst case is 1901.77. Hence, with respect to the result on the 3-fault faulty versions of these programs, a significant observation was made. It

was observed that in all but the best case of *flex*, the proposed approach is more effective than MSeer in all programs faulty versions.

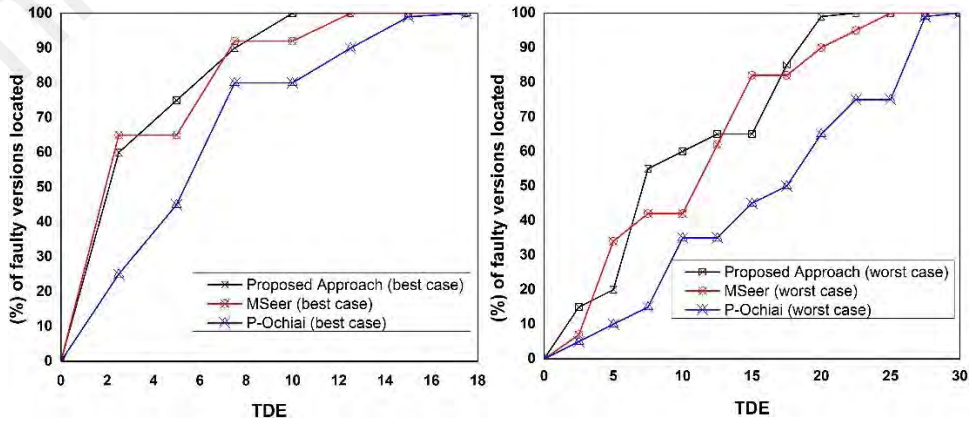
Table 5.18: Average number of statements examined using the proposed approach, MSeer, and P-Ochiai (3-fault versions)

	Proposed Approach		MSeer		P-Ochiai	
	Best case	Worst case	Best case	Worst case	Best case	Worst case
gzip	25.66	150.14	41.77	174.57	80.63	254.19
flex	30.18	101.40	23.30	111.40	110.03	243.63
grep	450.39	888.89	492.93	948.47	512.15	1901.77

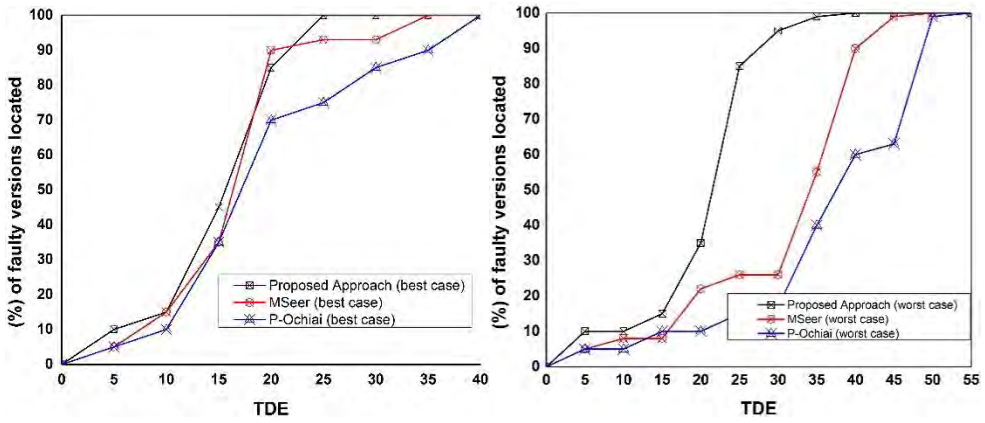
In all, the proposed approach is more effective because it examines fewer statements than MSeer and P-Ochiai in most cases. The difference between the proposed approach and MSeer is highly significant in some cases, for example, in the best case of *gzip*. In all cases, P-Ochiai is less effective, and the difference in comparison with other approaches is very significant. For instance, it was observed that the average number of statements to be examined in the best case of *flex* is 30.18 for the proposed approach and 110.03 for P-Ochiai in the best case. This clearly indicates that the proposed approach is much more effective than P-Ochiai. Another significant point worth noting is that in some cases, the effectiveness difference between the proposed approach and MSeer is insignificant. For example, in the worst case of *flex* where the proposed approach on average examined 101.40 statements in the worst case, and MSeer examined 111.40 in the worst case. Another significant observation is that in some cases, MSeer performs better than the proposed approach. For example, in the best case of *flex* where the proposed approach on average examined 30.18 statements in the best case, and MSeer examined 23.30 in the best case. Therefore, by examining lesser number of statements of *flex* than the proposed approach in the best case scenario, MSeer is the most effective in the scenario.

(m) ***Total developer expense (TDE)***

Next, the result of the cross-comparison between the proposed approach, MSeer, and P-Ochiai with respect to TDE score of all faulty versions is presented. Figure 5.21 presents the 3-fault versions of *gzip* and *grep* in the best and worst cases. For instance, in part (a) and part (b), by examining less than 10% of the program code, the proposed approach can locate 100% of the faulty versions in the best case and 60% in the worst case. MSeer (the second best approach) can only locate 92% in the best case and 42% in the worst case when examining the same amount of program code. For P-Ochiai (the third best), by examining the same amount of program code (less than 10%), it is 80% (best case) and 35% (worst case) to locate the faulty versions. Therefore, the proposed approach performs better than MSeer and P-Ochiai. In part (c) and (d) (*grep* 3-fault versions), the curves show that by examining less than 20% of the code, 85% of the faulty versions are located by the proposed approach in the best case and 35% in the worst case. For MSeer and P-Ochiai, these percentages are 90% and 70% in the best case, and 20% and 10% in the worst case, respectively. The results in Figure 5.20 and Figure 5.21 suggest that the proposed approach is convincingly the most effective in comparison to the best and worst cases of the comparative baseline approaches (MSeer and P-Ochiai).



(a) best case of gzip 3-fault versions (b) worst case of gzip 3-fault versions



(c) best case of grep 3-fault versions (d) worst case of grep 3-fault versions

Figure 5.21: TDE score-based comparison between the proposed approach with P-Ochiai and MSeer on gzip and grep (3-fault versions).

Based on the results in Table 5.18 and Figure 5.21, the proposed approach emerges to be the most effective in comparison to MSeer and P-Ochiai. Hence, with respect to the third evaluation metric, Table 5.19 gives the effectiveness comparison of the proposed approach in terms of the Wilcoxon signed-rank test on the 3-fault versions of *gzip, flex,* and *grep* programs. If the null hypothesis is accepted, the confidence level will be given as 00.00%. The cell with a black background in Table 5.19 is the cell where the null hypothesis is accepted.

Table 5.19: The confidence with which it can be claimed that the proposed approach is more effective than MSeer and P-Ochiai (best & worst cases) (3-fault versions)

	MSeer		P-Ochiai	
	Best case	Worst case	Best case	Worst case
gzip	93.80%	95.91%	98.19%	99.04%
flex	00.00%	90.00%	98.75%	99.30%
grep	97.65%	98.33%	98.39%	99.91%

For instance, for the best and worst cases of P-Ochiai on all programs, the confidence level to accept the alternative hypothesis is higher than 98%. Furthermore, for MSeer, the confidence to accept the alternative hypothesis is greater than 90% in most cases. However, for the best case of *flex* on MSeer, the null hypothesis is accepted, meaning that MSeer examined a fewer number of statements than the proposed approach which makes it more effective in this scenario. Therefore, because MSeer examined fewer statements than the proposed approach in the best case of *flex*, the null hypothesis is accepted and the confidence level is given as 00.00% in Table 5.19.

5.4.3. Distance Metrics

Distance metrics play a critical and important role to achieve a good tests clustering result, which measures the distance between failed tests or program statements (in the proposed approach context). In this section, the three distance metrics used by the proposed approach and the remaining two approaches (MSeer and P-Ochiai) were investigated. For the proposed approach, the edge-betweenness distance is used, MSeer used the revised Kendall tau distance, and P-Ochiai used the Euclidian distance metric. The effectiveness of these approaches using these distance metrics were compared. In Table 5.20, the average number of statements examined on 25 versions with 5-fault for *gzip, grep,* and *flex* are highlighted. Using the proposed approach, the average number of statements examined for *flex* is 95.83 (best case) and 186.99 (worst case). For MSeer and P-Ochiai, they are 104.67 and 230.06 (best cases), and 195.80 and 322.15 (worst cases), respectively. However, it was observed that in some cases, MSeer examined fewer statements. For example, in the best case of *grep* is 612.11 for the proposed approach and 598.80 for MSeer. Even though the difference is slightly small in some case between the

proposed approach and MSeer, the result clearly shows that the edge-betweenness distance used by the proposed approach is more effective.

Table 5.20: Average number of statements examined using the proposed approach, MSeer, and P-Ochiai (5-fault versions)

	Proposed Approach		MSeer		P-Ochiai	
	Best case	Worst case	Best case	Worst case	Best case	Worst case
Gzip	77.18	310.11	80.00	318.27	260.03	570.35
grep	612.11	1599.14	598.80	1644.77	621.34	2609.10
flex	95.83	186.99	104.67	195.80	230.06	322.15

5.4.4. Result Summary

For the results in this section, the following observations were made:

- Overall, based on the average number of statements examined by both the proposed approach and P-Ochiai in Table 5.14 and Table 15, the proposed approach is the most effective where in most cases, fewer statements were examined to locate all the faulty versions than P-Ochiai. For instance, it was observed that the average number of statements examined by the proposed approach on 2-fault faulty versions of *flex* is 9.08 (best case), and 62.18 (worst case). On the other hand, for P-Ochiai, it is 35.23 (best case), and 161.08 (worst case). Additionally, the proposed approach is still the most effective in terms of TDE score. However, it is worth highlighting that in some cases, the TDE score difference between the two approaches is not much. For example, with respect to *flex* program part (e) in the best case (Figure 5.20), using the proposed approach, all the faulty versions can be located by examining less than 10% of the code, and using P-Ochiai, all the faulty versions can be located by examining less than 12.5% of the program code.
- Furthermore, with respect to the 3-fault faulty versions of *flex*, it was observed that, in all but the best case of *flex*, the proposed approach is more effective,

however, MSeer on average can locate all the faulty versions by examining only 23.30 statements (best case), and 30.18 (best case) using the proposed approach. Another significant point worth noting is that in some cases, the effectiveness difference between the proposed approach and MSeer is insignificant. For example, in the worst case of *flex* where the proposed approach on average examined 101.40 statements (worst case), and MSeer examined 111.40 (worst case).

- In all program faulty versions of *gzip* and *grep* (Figure 5.21), the proposed approach is convincingly the most effective in the best and worst cases in comparison with MSeer and P-Ochiai approaches. For instance, in part (a) and part (b) of *gzip*, by examining less than 10% of the program code, the proposed approach can locate all the faulty versions in the best case and 60% (worst case). MSeer (the second best approach) can only locate 92% (best case) and 42% (worst case). For P-Ochiai (the third best), by examining the same amount of program code (less than 10%), it is 80% (best case) and 35% (worst case) to locate the faulty versions. Therefore, the proposed approach performs better than MSeer and P-Ochiai.

5.5. Chapter Summary

This chapter has presented and discussed the results of the four different experiments in this research. The investigative study on the claimed problematic parallel debugging approach, the multiple fault localization based on complex network theory (FLCN-M), the single fault localization based on complex network theory (FLCN-S), and the community-based fault isolation approach were evaluated on several single-fault and multiple-fault subject programs to access their effectiveness in terms of localization effectiveness. Cross-comparisons between the proposed fault localization techniques and

approach with the existing baseline techniques and approaches were conducted. In conclusion, the results in this chapter clearly shown that the proposed fault localization techniques and approach surpassed the existing baseline techniques and approaches in terms of locating faults in both single-fault and multiple-fault programs. The next chapter concludes the research by summarizing the research findings, highlighting the limitations of the research, presenting the research contributions, and giving recommendations for future work.

CHAPTER 6: CONCLUSION

This chapter concludes the current research by presenting: (i) the summary of the key findings in relation to the research objectives, (ii) the core research contributions, (iii) the main limitations of the study, and (iv) the future research directions of the study.

6.1. Summary of Findings in Relation to the Research Objectives

This research aims at proposing two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), to improve localization effectiveness in programs with single and multiple faults and aid developers to localize multiple faults simultaneously in a single diagnosis rank list. Furthermore, the research also aims at proposing a new community-based fault isolation approach to aid in the effective isolation and localization of multiple faults simultaneously in parallel. The two proposed fault localization techniques and the proposed approach were evaluated on several single-fault and multiple-fault subject programs in comparison with the existing baseline techniques and approaches to access their effectiveness in terms of localization effectiveness.

To develop the proposed fault localization techniques and approach, four research objectives as presented in Chapter 1, Section 1.4 have been formulated and fulfilled. As a result, it helps in developing the research methodology of this work. The research objectives for this work are as follows:

(1) To investigate the existing parallel debugging approach used in localizing multiple faults in terms of localization effectiveness in comparison with other debugging approaches.

(2) To propose two novel fault localization techniques for single-fault and multiple-fault programs based on complex network theory.

(3) To propose a new community-based fault isolation approach to aid in the effective isolation and localization of multiple faults simultaneously in parallel.

(4) To evaluate the proposed fault localization techniques and the proposed approach by comparing them with the baseline techniques and approaches in terms of localization effectiveness.

The ultimate goal of this research is to fulfill the above research objectives. Therefore, the following points establish that each of the research objectives was fulfilled by this research.

- To investigate the existing parallel debugging approach used in localizing multiple faults in terms of localization effectiveness in comparison with other debugging approaches.

This research objective is achieved by reviewing the literature on software fault localization as discussed in Chapter two. The existing studies that utilized different debugging approaches (i.e. OBA debugging approach and parallel debugging approach) to localize multiple faults were reviewed. The objective also investigates the claims of the existing works that a parallel debugging approach that performs clustering on failed tests based on their execution profile similarity, estimates the number of clusters based on the number of failed tests, and utilizes a distance metric like Euclidean distance to measure the *due-to* relationship between failed test cases, is problematic and inappropriate to use for the isolation and localization of multiple faults. The methodology

of this research work can be found in Chapter 3, Section 3.2. The findings show that even though the claimed problematic parallel debugging approach is effective and better in comparison with the OBA debugging approach, yet it is not as effective when compared with the state-of-the-art MSeer parallel debugging approach. Furthermore, it was observed that estimating the number of clusters based on the number of failed test cases as highlighted in Chapter 3, Section 3.2.2 is indeed not appropriate because there is no clear correlation between the number of failed test cases and the number of faults in a given program. Therefore, many redundant clusters can be generated that do not target faults but will increase the time and effort for a developer to look for faults. Therefore, the result affirms the previous studies claiming that performing clustering on failed tests based on their execution profile similarity, estimating the number of clusters based on the number of failed tests, and the utilization of metrics like Euclidean distance to measure the *due-to* relationship between failed test cases is problematic and inappropriate, which indeed reduces the effectiveness of a parallel debugging approach in localizing multiple faults.

- To propose two novel fault localization techniques for single-fault and multiple-fault programs based on complex network theory.

This objective was achieved by providing two novel fault localization techniques, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S). These techniques were proposed to improve localization effectiveness in programs with both single and multiple faults, and to aid developers to localize multiple faults simultaneously in a single diagnosis rank list. The proposed techniques rank statements based on their behavioral abnormalities and distance between statements in both passed and failed tests execution. Two graph-based centrality measures, namely degree centrality and closeness centrality

were adopted and used for fault diagnosis, and a new ranking formula was proposed to calculate the suspiciousness of program statements. The difference between the two techniques is that the former (FLCN-M) uses both passed and failed tests execution for fault localization, while the latter (FLCN-S) uses only failed tests execution for fault localization. The methodology of this research work can be found in Chapter 3, Section 3.3 and Section 3.4. The results have shown that the proposed techniques are significantly better in terms of fault localization effectiveness in both single-fault and multiple-fault programs when compared with the existing baseline fault localization techniques.

- To propose a new community-based fault isolation approach to aid in the effective isolation and localization of multiple faults simultaneously in parallel.

This objective was achieved by providing a new community-based fault isolation approach that makes use of a divisive network community clustering algorithm to aid in the isolation and localization of multiple faults simultaneously in parallel. This approach is applied in a scenario where a developer has checked 70% of the program statements using the proposed FLCN-M technique but cannot fully localize all the multiple faults in a single diagnosis rank list. In this case, instead of utilizing the OBA debugging approach, the community-based fault isolation approach will be utilised for a more efficient and effective localization of faults. The methodology of this research work can be found in Chapter 3, Section 3.5. The result has demonstrated that the proposed approach performs significantly better in terms of localization effectiveness in comparison with the claimed problematic parallel debugging approach and MSeer parallel debugging approach in locating multiple faults simultaneously in parallel.

- To evaluate the proposed fault localization techniques and the proposed approach by comparing them with the baseline techniques and approaches in terms of localization effectiveness.

The investigative study of the claimed problematic parallel debugging approach, the two novel fault localization techniques based on complex network theory, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), and the new community-based fault isolation approach were all evaluated on various single-fault and multiple-fault subject programs with six evaluation metrics as presented and detailed in Chapter 4. The aim is to access the localization effectiveness of the claimed problematic parallel debugging approach, the two proposed fault localization techniques, and the proposed approach using six evaluation metrics in comparison to other baseline techniques and approaches. However, to evaluate FLCN-M on localizing multiple faults in a single diagnosis rank list, a new generic evaluation metric named incremental developer expense (IDE) was proposed as presented in Chapter 4, Section 4.2.2.2. Due to the lack of this kind of evaluation metric in the literature, this metric was proposed to help developers to evaluate the effectiveness of localizing multiple faults simultaneously in a single diagnosis rank list. The metric is formulated to be generic and can be utilized by any fault localization techniques that localize faults in a single diagnosis rank list. The experimental results of this research on several single-fault and multiple-fault subject programs have shown that the claimed problematic parallel debugging approach is indeed problematic and contributes to the reduction in effectiveness of a parallel debugging approach in localizing multiple faults. Furthermore, based on the results in Chapter 5, the two proposed techniques and the proposed approach have shown to be more effective in identifying the locations of faults in comparison with the baseline techniques and approaches in the field of study.

6.2. Research Contributions

There are several contributions that have been gained from this research. The following are the key contributions of this research.

1. An investigative study of the claimed problematic parallel debugging approach that makes use of *k*-means clustering algorithm with Euclidean distance metric in terms of localizing multiple faults effectively in comparison with two other baseline debugging approaches, namely OBA debugging approach and MSeer parallel debugging approach.
2. A novel multiple fault localization technique that aids developers to effectively localize multiple faults simultaneously in a single diagnosis rank list.
3. A novel single fault localization technique that aids developers in localizing a single fault effectively.
4. A new community-based fault isolation approach that makes use of a divisive network community clustering algorithm to aid in the isolation and localization of multiple faults simultaneously in parallel.
5. A community weighting and selection process which aids in the selection and prioritization of *fault-focused* communities for effective simultaneous localization of faults in parallel.
6. A new generic evaluation metric named incremental developer expense (IDE) for the evaluation of FLCN-M fault localization technique on localizing multiple faults simultaneously in a single diagnosis rank list.
7. A comprehensive evaluation of the claimed problematic parallel debugging approach, the two proposed techniques, and the proposed approach on several single-fault and multiple-fault subject programs. Several baseline fault localization techniques and approaches have been used for cross-comparisons.

6.3. Limitations of the Study

In this thesis, one of the main limitations of the experiments is the exclusive utilization of subject programs that are of C programming language. Therefore, results generalization is quite limited due to this. Hence, more diverse programs in terms of language should be considered. Secondly, most of the programs utilized are somewhat considered to be either small or medium-sized with maximum lines of code of 13,892. Even though the UNIX real-life utility programs used in the experiments (*gzip, sed, flex,* and *grep*) are considerably larger-sized programs, they are still not very large due to the current diversity of existing software (in size and complexity). Recently, the software fault localization research domain is changing whereby experiments on much larger datasets are more generally preferred for better generalization.

Moreover, in most of the experiments in this research, artificial faults were used to create multiple-fault versions containing many faults where mutation-based injection technique is used for multiple-fault versions generation. Even though mutation-based faults can be used to simulate realistic faults and provide reliable and trustworthy results for testing and debugging experiments, more experiments on real programs with real-world faults are still necessary. This is because it is well known that existing fault localization techniques' performance varies when real faults reside in a program. Lastly, despite the improvement recorded in terms of localization effectiveness in the proposed community-based fault isolation approach, fault isolation accuracy is not optimal and needs improvement. Therefore, an improvement to the algorithm will be of great advantage to the software developer to lead to a better software quality.

6.4. Future Research Directions

Despite much attention researchers have given to software fault localization research domain, several research challenges exist that need more attention by current researchers. To improve results generality in software fault localization research and in the context of this thesis work, more experiments are needed on larger datasets with real faults. Recently, Siemens test suite dataset is not considered sufficient anymore despite being one of the most utilized datasets in software fault localization research domain. Therefore, adopting a more realistic and larger dataset is vital to help in generalizing the result of any study in software fault localization.

On the same note, further work on more diverse (in terms of language) subject programs of a different programming language such as Java and multilingual programs can be considered for further research. Furthermore, centrality measures used by the proposed techniques such as degree centrality and closeness centrality have proven to be effective, other centrality measures such as betweenness centrality, eigenvector centrality and so on, maybe explored in the proposed techniques to further improve fault localization effectiveness. Lastly, more work will be done to improve the divisive network community clustering algorithm used for the community-based fault isolation approach by adding a community estimation step to limit the number of communities produced so as to improve accuracy and further help in reducing the time to produce a failure-free program.

6.5. Final Words

Due to the complex relationship between fault and failure specifically in the existence of multiple faults, existing fault localization techniques find it hard to localize multiple faults simultaneously, whereby test cases that failed in the existence of single fault could pass in the existence of multiple faults, and a test case that passed in the presence of multiple faults could fail when a single fault is active. The simultaneous localization of multiple faults with good effectiveness has remained a prominent research problem in the field of software fault localization. To optimally solve the problem, firstly, a literature review and an investigative study were conducted that aid in achieving the first research objective. The conducted literature review and the investigative study helped in the development of the research methodology of this work. Next, two novel fault localization techniques, namely multiple fault localization based on complex network theory (FLCN-M) and single fault localization based on complex network theory (FLCN-S), were proposed to facilitate in achieving the second research objective. To fulfil the third research objective, a new community-based fault isolation approach was proposed. Finally, based on the experiments in this work, the localization effectiveness of the two proposed fault localization techniques and the proposed approach was empirically evaluated to help in fulfilling the last research objective of the study.

REFERENCES

Abreu, R., & van Gemund, A. J. (2009). *A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis.* Paper presented at the Eight Symposium on Abstraction, Reformation, and Approximation (SARA), (pp. 1-8).

Abreu, R., & Zoeteweij, P. (2006). *An evaluation of similarity coefficients for software fault localization.* Paper presented at the 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). IEEE, (pp. 39-46).

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2007). *On the accuracy of spectrum-based fault localization.* Paper presented at the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. (TAICPART-MUTATION) IEEE, (pp. 89-98).

Abreu, R., Zoeteweij, P., & van Gemund, A. J. (2008). *A dynamic modeling approach to software multiple-fault localization.* Paper presented at the Proceedings of the 19th International Workshop on Principles of Diagnosis, (pp. 7-14).

Abreu, R., Zoeteweij, P., & van Gemund, A. J. (2009a). *Localizing software faults simultaneously.* Paper presented at the 2009 Ninth International Conference on Quality Software (pp. 367-376).

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2009b). *Spectrum-based multiple fault localization.* Paper presented at the 24th IEEE/ACM International Conference on Automated Software Engineering ASE'09, (pp. 88-99).

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2011). Simultaneous debugging of software faults. *Journal of systems and software, 84*(4), 573-586.

Agrawal, H., De Millo, R. A., & Spafford, E. H. (1991). An execution-backtracking approach to debugging. *IEEE Software, 8*(3), 21-26.

Agrawal, H., DeMillo, R. A., & Spafford, E. H. (1990). *Efficient Debugging with Slicing and Backtracking.*

Agrawal, H., DeMillo, R. A., & Spafford, E. H. (1993). Debugging with dynamic slicing and backtracking. *Software: Practice and Experience, 23*(6), 589-616.

Agrawal, H., & Horgan, J. R. (1990). *Dynamic program slicing.* Paper presented at the ACM SIGPLAN Notices, (vol. 25, no. 6, pp. 246-256).

Agrawal, H., Horgan, J. R., London, S., & Wong, W. E. (1995). *Fault localization using execution slices and dataflow tests.* Paper presented at the Sixth International Symposium on Software Reliability Engineering, ISSRE'95. (pp. 143-151).

Albert, R., & Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Reviews of modern physics, 74*(1), 47.

Andrews, J. H., Briand, L. C., & Labiche, Y. (2005). *Is mutation an appropriate tool for testing experiments?* Paper presented at the Proceedings of the 27th international conference on Software engineering, (pp. 402-411).

Andrews, J. H., Briand, L. C., Labiche, Y., & Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE transactions on Software Engineering* (8), 608-624.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing, 1*(1), 11-33.

Baah, G. K., Podgurski, A., & Harrold, M. J. (2010). The probabilistic program dependence graph and its application to fault diagnosis. *IEEE transactions on Software Engineering, 36*(4), 528-545.

Baudry, B., Fleurey, F., & Le Traon, Y. (2006). *Improving test suites for efficient fault localization.* Paper presented at the Proceedings - International Conference on Software Engineering, (pp. 82-91).

Bibby, J., Kent, J., & Mardia, K. (1979). Multivariate analysis: Academic Press, London.

Binkley, D., Gold, N., & Harman, M. (2007). An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM), 16*(2), 8.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment, 2008*(10), P10008.

Borgatti, S. P. (2005). Centrality and network flow. *Social networks, 27*(1), 55-71.

Bornholdt, S., & Schuster, H. G. (2006). *Handbook of graphs and networks: from the genome to the internet*: John Wiley & Sons.

Briand, L. C., Labiche, Y., & Liu, X. (2007). *Using machine learning to support debugging with tarantula.* Paper presented at the 18th IEEE International Symposium on Software Reliability, (ISSRE'07), (pp. 137-146).

Browne, M., & Ghidary, S. S. (2003). *Convolutional neural networks for image processing: an application in robot vision.* Paper presented at the Australasian Joint Conference on Artificial Intelligence., (pp. 641-652)

Bullmore, E., & Sporns, O. (2009). Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience, 10*(3), 186-198.

Cai, K.-Y., & Yin, B.-B. (2009). Software execution processes as an evolving complex network. *Information Sciences, 179*(12), 1903-1928.

Chen, D., Lü, L., Shang, M.-S., Zhang, Y.-C., & Zhou, T. (2012). Identifying influential nodes in complex networks. *Physica a: Statistical mechanics and its applications, 391*(4), 1777-1787.

Cheng, Y., & Suthers, D. (2011). SOCIAL NETWORK ANALYSIS—CENTRALITY MEASURES.

Choi, S.-S., Cha, S.-H., & Tappert, C. C. (2010). A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics, 8*(1), 43-48.

Chong, C. Y., & Lee, S. P. (2015). Analyzing maintainability and reliability of object-oriented software using weighted complex network. *Journal of systems and software, 110*, 28-53.

Cleve, H., & Zeller, A. (2005). *Locating causes of program failures.* Paper presented at the 27th International Conference on Software Engineering, (ICSE'05), (pp. 342-351)

Collofello, J. S., & Woodfield, S. N. (1989). Evaluating the effectiveness of reliability-assurance techniques. *Journal of systems and software, 9*(3), 191-195.

Cousin, L. D. (1986). Towards automatic software fault location through decision-to-decision path analysis.

de Souza, H. A., Mutti, D., Chaim, M. L., & Kon, F. (2018). Contextualizing spectrum-based fault localization. *Information and software technology, 94*, 245-261.

Dean, B. C., Pressly, W. B., Malloy, B. A., & Whitley, A. A. (2009). *A linear programming approach for automated localization of multiple faults.* Paper presented at the Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, (pp. 640-644).

Debroy, V., & Wong, W. E. (2009). *Insights on fault interference for programs with multiple bugs.* Paper presented at the 20th International Symposium on Software Reliability Engineering, (ISSRE'09), (pp. 165-174)

DeMillo, R. A., Pan, H., & Spafford, E. H. (1996). *Critical slicing for software fault localization.* Paper presented at the ACM SIGSOFT Software Engineering Notes, (vol. 21, no. 3, pp. 121-134).

DeMillo, R. A., Pan, H., & Spafford, E. H. (1997). *Failure and fault analysis for software debugging.* Paper presented at the Proceedings - IEEE Computer Society's International Computer Software and Applications Conference, (COMPSAC'97), (pp. 515-521).

Denmat, T., Ducassé, M., & Ridoux, O. (2005). *Data mining and cross-checking of execution traces: a re-interpretation of jones, harrold and stasko test information.* Paper presented at the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, (ASE'05), (pp. 396-399).

Despain, A. M., & Patterson, D. A. (1978). *X-Tree: A tree structured multi-processor computer architecture.* Paper presented at the Proceedings of the 5th annual symposium on Computer architecture, (pp. 144-151).

DiGiuseppe, N., & Jones, J. A. (2011a). *Fault interaction and its repercussions.* Paper presented at the 27th IEEE International Conference on Software Maintenance (ICSM), (pp. 3-12).

DiGiuseppe, N., & Jones, J. A. (2011b). *On the influence of multiple faults on coverage-based fault localization.* Paper presented at the Proceedings of the 2011 international symposium on software testing and analysis, (pp. 210-220).

DiGiuseppe, N., & Jones, J. A. (2012a). *Concept-based failure clustering.* Paper presented at the Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, (pp. 29).

DiGiuseppe, N., & Jones, J. A. (2012b). *Software behavior and failure clustering: An empirical study of fault causality.* Paper presented at the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, (pp. 191-200).

DiGiuseppe, N., & Jones, J. A. (2015). Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering, 20*(4), 928-967.

Do, H., Elbaum, S., & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering, 10*(4), 405-435.

Dorogovtsev, S., & Mendes, J. (2003). Evolution of Networks: from biological nets to the Internet and WWW Oxford U. Press: Oxford.

Freeman, L. (2004). The development of social network analysis. *A Study in the Sociology of Science*.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, 35-41.

Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social networks, 1*(3), 215-239.

Gao, C., Wei, D., Hu, Y., Mahadevan, S., & Deng, Y. (2013). A modified evidential methodology of identifying influential nodes in weighted networks. *Physica a: Statistical mechanics and its applications, 392*(21), 5490-5500.

Gao, R., & Wong, W. E. (2017). MSeer-An Advanced Technique for Locating Multiple Bugs in Parallel. *IEEE transactions on Software Engineering*.

Girvan, M., & Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the national academy of sciences, 99*(12), 7821-7826.

Gong, C., Zheng, Z., Zhang, Y., Zhang, Z., & Xue, Y. (2012). -. Paper presented at the 2012 19th Asia-Pacific Software Engineering Conference.

Gupta, N., He, H., Zhang, X., & Gupta, R. (2005). *Locating faulty code using failure-inducing chops.* Paper presented at the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, (pp. 263-272).

Gyimóthy, T., Beszédes, Á., & Forgács, I. (1999). *An efficient relevant slicing method for debugging.* Paper presented at the Software Engineering—ESEC/FSE'99, (pp. 303-321).

Harrold, M. J., Rothermel, G., Wu, R., & Yi, L. (1998). *An empirical investigation of program spectra.* Paper presented at the ACM SIGPLAN Notices, (vol. 33, no. 7, pp. 83-90).

Hartigan, J. A., & Wong, M. A. (1979). Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics), 28*(1), 100-108.

Hennessy, J. (1982). Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS), 4*(3), 323-344.

Högerle, W., Steimann, F., & Frenkel, M. (2014). *More Debugging in Parallel.* Paper presented at the 2014 IEEE 25th International Symposium on Software Reliability Engineering, (pp. 133-143).

Huang, Y., Wu, J., Feng, Y., Chen, Z., & Zhao, Z. (2013). *An empirical study on clustering for isolating bugs in fault localization.* Paper presented at the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), (pp. 138-143).

Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). *Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria.* Paper presented at the Proceedings of the 16th international conference on Software engineering, (pp. 191-200).

Jeffrey, D., Gupta, N., & Gupta, R. (2009). *Effective and efficient localization of multiple faults using value replacement.* Paper presented at the IEEE International Conference on Software Maintenance, (ICSM'09), (pp. 221-230).

Jones, J. A., Bowring, J. F., & Harrold, M. J. (2007). *Debugging in parallel.* Paper presented at the Proceedings of the 2007 international symposium on Software testing and analysis, (pp. 16-26).

Jones, J. A., & Harrold, M. J. (2005). *Empirical evaluation of the tarantula automatic fault-localization technique.* Paper presented at the 20th IEEE/ACM International Conference on Automated Software Engineering, (ASE'05), (pp. 273-282).

Jones, J. A., Harrold, M. J., & Stasko, J. (2002). *Visualization of test information to assist fault localization.* Paper presented at the Proceedings - International Conference on Software Engineering, (pp. 467-477).

Ju, X., Jiang, S., Chen, X., Wang, X., Zhang, Y., & Cao, H. (2014). HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of systems and software, 90*, 3-17.

Kim, J., Kim, J., & Lee, E. (2018). *A novel variable-centric fault localization technique.* Paper presented at the Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, (pp. 252-253).

Kiss, Á., Jász, J., & Gyimóthy, T. (2005). Using dynamic information in the interprocedural static slicing of binary executables. *Software Quality Journal, 13*(3), 227-245.

Könighofer, R., & Bloem, R. (2011). *Automated error localization and correction for imperative programs.* Paper presented at the Formal Methods in Computer-Aided Design (FMCAD), 2011.

Korel, B. (1988). PELAS-program error-locating assistant system. *IEEE transactions on Software Engineering, 14*(9), 1253-1260.

Korel, B., & Laski, J. (1988). Dynamic program slicing. *Information Processing Letters, 29*(3), 155-163. doi:10.1016/0020-0190(88)90054-3

Korel, B., & Laski, J. (1988). *STAD-A system for testing and debugging: User perspective.* Paper presented at the Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, (pp. 13-20).

Kung, J., Kim, D., & Mukhopadhyay, S. (2015). On the impact of energy-accuracy tradeoff in a digital cellular neural network for image processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 34*(7), 1070-1081.

Kusumoto, S., Nishimatsu, A., Nishie, K., & Inoue, K. (2002). Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering, 7*(1), 49-76. doi:10.1023/A:1014823126938

Lamraoui, S.-M., & Nakajima, S. (2016). A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs. *Journal of Information Processing, 24*(1), 88-98.

Landsberg, D., Sun, Y., & Kroening, D. (2018). *Optimising spectrum based fault localization for single fault programs using specifications.* Paper presented at the International Conference on Fundamental Approaches to Software Engineering, (pp. 246-263).

Le, T.-D. B., Thung, F., & Lo, D. (2013). *Theory and practice, do they match? a case with spectrum-based fault localization.* Paper presented at the 29th IEEE International Conference on Software Maintenance (ICSM'13), (pp. 380-383).

Lee, J., Kim, J., & Lee, E. (2016). *Enhanced Fault Localization by Weighting Test Cases with Multiple Faults.* Paper presented at the Proceedings of the International Conference on Software Engineering Research and Practice (SERP), (pp. 116).

Leiserson, C. E., Rivest, R. L., Stein, C., & Cormen, T. H. (2001). Introduction to algorithms: The MIT press Cambridge.

Li, D., Han, Y., & Hu, J. (2008). *Complex network thinking in software engineering.* Paper presented at the International Conference on Computer Science and Software Engineering, (pp. 264-268).

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable statistical bug isolation. *ACM SIGPLAN Notices, 40*(6), 15-26.

Liu, A., Li, L., & Luo, J. (2015). *Automated Program Debugging for Multiple Bugs Based on Semantic Analysis.* Paper presented at the International Workshop on Structured Object-Oriented Formal Language and Method, (pp. 86-100).

Liu, B., Nejati, S., Briand, L., & Bruckmann, T. (2016). *Localizing Multiple Faults in Simulink Models.* Paper presented at the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), (pp. 146-156).

Liu, C., Fei, L., Yan, X., Han, J., & Midkiff, S. P. (2006). Statistical debugging: A hypothesis testing-based approach. *IEEE transactions on Software Engineering, 32*(10), 831-848.

Liu, C., Zhang, X., & Han, J. (2008). A systematic study of failure proximity. *IEEE transactions on Software Engineering, 34*(6), 826-843.

Lyle, R. (1987). *Automatic program bug location by program slicing.* Paper presented at the Proceedings 2nd International Conference on Computers and Applications, (pp. 877-883).

Ma, Y., He, K., & Du, D. (2005). *A qualitative method for measuring the structural complexity of software systems based on complex networks.* Paper presented at the12th Asia-Pacific Software Engineering Conference, (APSEC'05), (pp. 7).

Mao, X., Lei, Y., Dai, Z., Qi, Y., & Wang, C. (2014). Slice-based statistical fault localization. *Journal of systems and software, 89*, 51-62.

Masri, W., & Assi, R. A. (2014). Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM), 23*(1), 8.

Mateis, C., Stumptner, M., & Wotawa, F. (2000). *Modeling Java programs for diagnosis.* Paper presented at the Proceedings of the 14th European Conference on Artificial Intelligence, (pp. 171-175).

Mayer, W., Abreu, R., Stumptner, M., & van Gemund, A. (2008). *Prioritising model-based debugging diagnostic reports.* Paper presented at the Proceedings of the 19th International Workshop on Principles of Diagnosis, (pp. 127-134).

Mayer, W., & Stumptner, M. (2002). *Modeling programs with unstructured control flow for debugging.* Paper presented at the Australian Joint Conference on Artificial Intelligence, (pp. 107-118).

Mayer, W., & Stumptner, M. (2004). *Approximate modeling for debugging of program loops.* Citeseer.

Mayer, W., Stumptner, M., & Wotawa, F. (2003). *Debugging program exceptions.* DX-03.

Myers, C. R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E, 68*(4), 046116.

Naish, L., Lee, H. J., & Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology (TOSEM), 20*(3), 11.

Namin, A. S. (2015). *Statistical Fault Localization Based on Importance Sampling.* Paper presented at the IEEE 14th International Conference on Machine Learning and Applications (ICMLA), (pp. 58-63).

Neelofar, N., Naish, L., Lee, J., & Ramamohanarao, K. (2017). Improving spectral-based fault localization using static analysis. *Software: Practice and Experience. 47(11) 1633-1655*.

Neelofar, N., Naish, L., & Ramamohanarao, K. (2018). Spectral-based fault localization using hyperbolic function. *Software: Practice and Experience, 48*(3), 641-664.

Newman, M. E. (2001). Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. *Physical Review E, 64*(1), 016132.

Newman, M. E., & Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E, 69*(2), 026113.

NIST. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing [Online]. Retrieved from http://www.abeacha.com/NIST_press_release_bugs_cost.html

Opsahl, T., & Panzarasa, P. (2009). Clustering in weighted networks. *Social networks, 31*(2), 155-163.

Ott, R. L., & Longnecker, M. T. (2015). *An introduction to statistical methods and data analysis*: Nelson Education.

Parsa, S., Vahidi-Asl, M., & Asadi-Aghbolaghi, M. (2014). Hierarchy-Debug: a scalable statistical technique for fault localization. *Software Quality Journal, 22*(3), 427-466.

Parsa, S., Vahidi-Asl, M., & Zareie, F. (2016). Statistical Based Slicing Method for Prioritizing Program Fault Relevant Statements. *Computing and Informatics, 34*(4), 823-857.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., . . . Keller, B. (2017). *Evaluating and improving fault localization.* Paper presented at the Proceedings of the 39th International Conference on Software Engineering, (pp. 609-620).

Perez, A., Abreu, R., & van Deursen, A. (2017). *A test-suite diagnosability metric for spectrum-based fault localization approaches.* Paper presented at the Proceedings of the 39th International Conference on Software Engineering, (654-664).

Renieres, M., & Reiss, S. P. (2003). *Fault localization with nearest neighbor queries.* Paper presented at the 18th IEEE International Conference on Automated Software Engineering, (pp. 30-39).

Reps, T., Ball, T., Das, M., & Larus, J. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem *Software Engineering—ESEC/FSE'97* (pp. 432-449): Springer.

Rosenblum, D. S. (1995). A practical approach to programming with assertions. *IEEE transactions on Software Engineering, 21*(1), 19-31.

Scott, J. (2000). Social network analysis: A handbook. Sage London. *2nd edtion*.

Shu, T., Ye, T., Ding, Z., & Xia, J. (2016). Fault localization based on statement frequency. *Information Sciences, 360*, 43-56.

Srivastav, M., Singh, Y., Gupta, C., & Chauhan, D. S. (2010). *Complexity estimation approach for debugging in parallel.* Paper presented at the 2010 Second International Conference on Computer Research and Development, (pp. 223-227).

Steimann, F., & Bertschler, M. (2009). *A simple coverage-based locator for multiple faults.* Paper presented at the International Conference on Software Testing Verification and Validation, ICST'09, (pp. 366-375).

Steimann, F., & Frenkel, M. (2012). *Improving coverage-based localization of multiple faults using algorithms from integer linear programming.* Paper presented at the 2012 IEEE 23rd International Symposium on Software Reliability Engineering, (pp. 121-130).

Strogatz, S. H. (2001). Exploring complex networks. *Nature, 410*(6825), 268-276.

Šubelj, L., & Bajec, M. (2011). Community structure of complex software systems: Analysis and applications. *Physica a: Statistical mechanics and its applications, 390*(16), 2968-2975.

Šubelj, L., & Bajec, M. (2012). *Software systems through complex networks science: Review, analysis and applications.* Paper presented at the Proceedings of the First International Workshop on Software Mining, (pp. 9-16).

Sun, X., Li, B., & Wen, W. (2013). *CLPS-MFL: Using Concept Lattice of Program Spectrum for Effective Multi-fault Localization.* Paper presented at the 2013 13th International Conference on Quality Software, (pp. 204-207).

Sun, X., Peng, X., Li, B., Li, B., & Wen, W. (2016). IPSETFUL: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra. *Frontiers of Computer Science, 10*(5), 812-831.

Tip, F., & Dinesh, T. (2001). A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology (TOSEM), 10*(1), 5-55.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies, 23*(5), 459-494.

Wang, X., & Liu, Y. (2016). Fault localization using disparities of dynamic invariants. *Journal of systems and software, 122*, 144-154.

Wang, Y., Gao, R., Chen, Z., Wong, W. E., & Luo, B. (2014). WAS: A weighted attribute-based strategy for cluster test selection. *Journal of systems and software, 98*, 44-58.

Wang, Y., Huang, Z., Fang, B., & Li, Y. (2018). Spectrum-Based Fault Localization via Enlarging Non-Fault Region to Improve Fault Absolute Ranking. *IEEE Access, 6*, 8925-8933.

Weeratunge, D., Zhang, X., Sumner, W. N., & Jagannathan, S. (2010). *Analyzing concurrency bugs using dual slicing.* Paper presented at the Proceedings of the 19th international symposium on Software testing and analysis, (pp. 253-264).

Wei, Z., & Han, B. (2013). *Multiple-Bug Oriented Fault Localization: A Parameter-Based Combination Approach.* Paper presented at the IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C), (pp. 125-130).

Weiser, M. (1984). {Program slicing.{IEEE Transactions on Software Engineering, SE-10 (4): 352 {357: July.

Weiser, M. D. (1979). Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.

Witten, I. H., & Frank, E. (1999). Data mining: practical machine learning tools and techniques with Java implementations: Morgan Kaufmann San Francisco.

Wong, E., Wei, T., Qi, Y., & Zhao, L. (2008). *A crosstab-based statistical method for effective fault localization.* Paper presented at the 1st International Conference on Software Testing, Verification, and Validation, (pp. 42-51).

Wong, W. E., Debroy, V., Gao, R. Z., & Li, Y. H. (2014). The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability, 63*(1), 290-308. doi:10.1109/tr.2013.2285319

Wong, W. E., Debroy, V., Golden, R., Xu, X., & Thuraisingham, B. (2012). Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability, 61*(1), 149-169.

Wong, W. E., Debroy, V., & Xu, D. (2012). Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42*(3), 378-396.

Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. *IEEE transactions on Software Engineering, 42*(8), 707-740.

Wong, W. E., & Qi, Y. (2009). BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION. *International Journal of Software Engineering and Knowledge Engineering, 19*(4), 573-597. doi:10.1142/s021819400900426x

Wotawa, F. (2010). *Fault localization based on dynamic slicing and hitting-set computation.* Paper presented at the 10th International Conference on Quality Software (QSIC), (161-170).

Wotawa, F., Nica, M., & Moraru, I. (2012). Automated debugging based on a constraint model of the program and a test case. *The journal of logic and algebraic programming, 81*(4), 390-407.

Wotawa, F., Stumptner, M., & Mayer, W. (2002). *Model-based debugging or how to diagnose programs automatically.* Paper presented at the International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, (pp. 746-757).

Xia, X., Gong, L., Le, T.-D. B., Lo, D., Jiang, L., & Zhang, H. (2016). Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *Automated Software Engineering, 23*(1), 43-75.

Xiaobo, Y., Bin, L., & Jianxing, L. (2017). *The Failure Behaviors of Multi-Faults Programs: An Empirical Study.* Paper presented at the Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on, (pp. 1-7).

Xie, X., Wong, W. E., Chen, T. Y., & Xu, B. (2013). Metamorphic slice: An application in spectrum-based fault localization. *Information and software technology, 55*(5), 866-879.

Xu, J., Zhang, Z., Chan, W. K., Tse, T., & Li, S. (2013). A general noise-reduction framework for fault localization of Java programs. *Information and software technology, 55*(5), 880-896.

Xue, X., & Namin, A. S. (2013). *How Significant is the Effect of Fault Interactions on Coverage-Based Fault Localizations?* Paper presented at the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, (pp. 113-122).

Yoo, S., Xie, X., Kuo, F.-C., Chen, T. Y., & Harman, M. (2014). No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *RN, 14*(14), 14.

You, Z., Qin, Z., & Zheng, Z. (2012). *Statistical fault localization using execution sequence.* Paper presented at the 2012 International Conference on Machine Learning and Cybernetics (ICMLC), (pp. 899-905).

Yu, Y., Jones, J. A., & Harrold, M. J. (2008). *An empirical study of the effects of test-suite reduction on fault localization.* Paper presented at the Proceedings of the 30th international conference on Software engineering, (pp. 201-210).

Zakari, A., Lawan, A. A., & Bekaroo, G. (2016). *A Hybrid Three-Phased Approach in Requirement Elicitation.* Paper presented at the International Conference on Emerging Trends in Electrical, Electronic and Communications Engineering, (pp. 331-340).

Zakari, A., Lee, S., Alam, K. A., & Ahmad, R. (2018). Software Fault Localization: A Systematic Mapping Study. *IET Software*.

Zakari, A., Lee, S. P., & Chong, C. Y. (2018). Simultaneous Localization of Software Faults Based on Complex Network Theory. *IEEE Access*.

Zeller, A. (2002). *Isolating cause-effect chains from computer programs.* Paper presented at the Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, (pp.).

Zhang, X.-Y., Zheng, Z., & Cai, K.-Y. (2018). Exploring the usefulness of unlabelled test cases in software fault localization. *Journal of systems and software, 136*, 278-290.

Zhang, X., Gupta, N., & Gupta, R. (2007). Locating faulty code by multiple points slicing. *Software: Practice and Experience, 37*(9), 935-961.

Zhang, X., Tallam, S., Gupta, N., & Gupta, R. (2007). *Towards locating execution omission errors.* Paper presented at the ACM SIGPLAN Notices, (pp. 415-424).

Zhang, Z., Chan, W. K., & Tse, T. (2012). Fault localization based only on failed runs. *Computer, 45*(6), 64-71.

Zhang, Z., Jiang, B., Chan, W. K., Tse, T., & Wang, X. (2010). Fault localization through evaluation sequences. *Journal of systems and software, 83*(2), 174-187.

Zhao, L., Zhang, Z., Wang, L., & Yin, X. (2013). A fault localization framework to alleviate the impact of execution similarity. *International Journal of Software Engineering and Knowledge Engineering, 23*(07), 963-998.

Zheng, A. X., Jordan, M. I., Liblit, B., & Aiken, A. (2003). *Statistical debugging of sampled programs.* Paper presented at the Advances in Neural Information Processing Systems, (pp. 603-610).

Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., & Aiken, A. (2006). *Statistical debugging: simultaneous identification of multiple bugs.* Paper presented at the Proceedings of the 23rd international conference on Machine learning, (pp. 1105-1112).

Zheng, W., Hu, D. S., & Wang, J. (2016). Fault Localization Analysis Based on Deep Neural Network. *Mathematical Problems in Engineering, 2016*. doi:Artn 1820454 10.1155/2016/1820454

Zheng, Y., Wang, Z., Fan, X., Chen, X., & Yang, Z. (2018). Localizing multiple software faults based on evolution algorithm. *Journal of systems and software, 139*, 107-123.

Zhu, L.-Z., Yin, B.-B., & Cai, K.-Y. (2011). *Software fault localization based on centrality measures.* Paper presented at the IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW), (pp. 37-42).

LIST OF PUBLICATIONS AND PAPERS PRESENTED

- **Journal Papers:**

(1) Zakari, A., Lee, S. P., & Chong, C. Y. (2018). Simultaneous Localization of Software Faults Based on Complex Network Theory. *IEEE Access*, *6*, 23990-24002. **(Published)**

(2) Zakari, A., Lee, S. P., Alam, K. A., & Ahmad, R. (2018). Software Fault Localization: A Systematic Mapping Study. *IET Software*. **(Published)**

(3) Zakari, A., & Lee, S. P., & Ibrahim, T. (2019). A Community-based Fault Isolation Approach for Effective Simultaneous Localization of Faults. *IEEE Access*. **(Published)**

(4) Zakari, A., & Lee, S. P. Parallel Debugging: An Investigative Study. *Journal of Software: Evolution and Process*. **(Minor Revision)**

(5) Zakari, A., Lee, S. P., & Ibrahim, T. Single Fault Localization Technique based on Failed Test Input. *Computers*. **(Major Revision)**

(6) Zakari, A., Lee, S. P., Bashir, R. P., & Bekaroo, G. Multiple Fault Localization of Software Programs: State-of-the-art, Issues, and Challenges. *Expert Systems with Applications*. **(Under Review)**

- **Conference Papers**

(1) Zakari, A., & Lee, S. P. (2017). Software Fault Localization: Issues and Limitations. *PGRES, FCSIT, University Malaya*. **(Published)**

(2) Zakari, A., & Lee, S. P. (2019). Simultaneous Isolation of Software Faults for Effective Fault Localization. *15th IEEE Colloquium on Signal Processing and its Applications* (CSPA'19). **(Published)**