

FEDERAL UNIVERSITY OF PARANÁ

JACKSON ANTONIO DO PRADO LIMA

A MULTI-ARMED BANDIT APPROACH FOR ENHANCING TEST CASE
PRIORITIZATION IN CONTINUOUS INTEGRATION ENVIRONMENTS

CURITIBA PR

2021

JACKSON ANTONIO DO PRADO LIMA

A MULTI-ARMED BANDIT APPROACH FOR ENHANCING TEST CASE
PRIORITIZATION IN CONTINUOUS INTEGRATION ENVIRONMENTS

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computer Science*.

Orientador: Silvia Regina Vergilio.

CURITIBA PR

2021

P896

Prado Lima, Jackson Antonio do

A Multi-Armed Bandit Approach for Enhancing Test Case
Prioritization in Continuous Integration environments [recurso
eletrônico] / Jackson Antonio do Prado Lima - Curitiba, 2021.

Tese (doutorado) - Programa de Pós-Graduação em Informática, Setor
de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Profa. Dra. Silvia Regina Vergilio

1. Ciência da Computação. 2. Software - Testes. 3. Algoritmos de
computador. I. Universidade Federal do Paraná. II. Vergilio, Silvia
Regina. III. Título.

CDD 005.1

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **JACKSON ANTONIO DO PRADO LIMA** intitulada: **A Multi-Armed Bandit Approach for Enhancing Test Case Prioritization in Continuous Integration environments**, sob orientação da Profa. Dra. SILVIA REGINA VERGILIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 10 de Dezembro de 2021.

Assinatura Eletrônica

13/12/2021 11:59:44.0

SILVIA REGINA VERGILIO

Presidente da Banca Examinadora

Assinatura Eletrônica

14/01/2022 12:03:08.0

ALESSANDRO FABRICIO GARCIA

Avaliador Externo (PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO
DE JANEIRO)

Assinatura Eletrônica

14/12/2021 15:05:55.0

BRENO MIRANDA

Avaliador Externo (UNIVERSIDADE FEDERAL DE PERNAMBUCO)

Assinatura Eletrônica

10/01/2022 10:31:40.0

MARIA CLAUDIA FIGUEIREDO PEREIRA EMER

Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO
PARANÁ)

Assinatura Eletrônica

14/12/2021 09:36:46.0

AURORA TRINIDAD RAMIREZ POZO

Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

To my wife, son, and parents...

ACKNOWLEDGEMENTS

This thesis had support from several people. Firstly, I would like to thank my parents, Marcio do Prado Lima and Sibila Brüskey do Prado Lima, for all support they have given me. Thanks for encouraging me to pursue my goals and to focus on my studies, mainly when my goals looked impossible you pushed me back to the right path. In this path, my sister Suelen do Prado Lima was on my side, protecting and taking care of me.

I would like to thank my wife, Jéssica Röpke do Prado Lima. This doctorate would not be possible without your love and patience. Due to the pandemic situation, you made the difference more than ever. Thank you for standing on my side all these years. Moreover, I would like to thank my son Miguel Augusto do Prado Lima. You bring me immense happiness in my heart. Thanks for making this end of doctorate more special than ever.

Special thanks to my advisor, Prof. Dr. Silvia Regina Vergilio. You believed in me. Beyond the excellent and outstanding supervision, and standing with me since the master's degree, you were my second mother. Indeed. In the moments that I would like to give up, you were there to listen to me and help me. Moreover, I will not forget our discussion of the works and your supervision style, they will be with me forever!

Thanks to Prof. Dr. Aurora Trinidad Ramirez Pozo from the C-Bio group at UFPR. You had great importance during my academic journey since the master's degree. In addition to the advice and guidance, you were a friend.

Thanks to all my friends from the GrEs and C-Bio research groups. You were my family during these almost seven years. We shared great moments together, and I learned a lot! Moreover, my gratitude to all other friends and family members who were by my side during these years. I am grateful for all.

Finally, I would like to GENI (Global Environment for Networking Innovation). I would be not able to conduct my experiments without the infrastructure provided.

This work is also supported by the Brazilian funding agency Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) under the grant: 88882.382199/2019-01.

RESUMO

A Integração Contínua (do inglês *Continuous Integration*, CI) é uma prática comum e amplamente adotada na indústria que permite a integração frequente de mudanças de software, tornando a evolução do software mais rápida e econômica. Em ambientes que adotam CI, o Teste de Regressão (do inglês *Regression Testing*, RT) é fundamental para assegurar que mudanças realizadas não afetaram negativamente o comportamento do sistema. No entanto, RT é uma tarefa cara. Para reduzir os custos do RT, o uso de técnicas de priorização de casos de teste (do inglês *Test Case Prioritization*, TCP) desempenha um papel importante. Essas técnicas visam a identificar a ordem para os casos de teste que maximiza objetivos específicos, como a detecção antecipada de falhas. Recentemente, muitos estudos surgiram no contexto de TCP para ambientes de CI (do inglês *Test Case Prioritization in Continuous Integration*, TCPCI), mas poucos estudos consideram particularidades destes ambientes, tais como restrições de tempo e a volatilidade dos casos de teste, ou seja, eles não consideram o ambiente dinâmico do ciclo de vida do software no qual novos casos de teste podem ser adicionados ou removidos (descontinuados) de um ciclo para outro. A volatilidade de casos de teste está relacionada ao dilema de Exploração versus Intensificação (do inglês *Exploration versus Exploitation*, EvE). Para resolver este dilema uma abordagem precisa balancear: i) a diversidade do conjunto de testes; e ii) a quantidade de novos casos de teste e testes que possuem alta probabilidade de revelar defeitos. Para lidar com isso, a maioria das abordagens usa, além do histórico de falhas, outras métricas que consideram instrumentação de código ou necessitam de informações adicionais, tais como a cobertura de testes. Contudo, manter as informações atualizadas pode ser difícil e consumir tempo, e não ser escalável devido ao orçamento de teste do ambiente de CI. Neste contexto, e para lidar apropriadamente com o problema de TCPCI, este trabalho apresenta uma abordagem baseada em problemas Multi-Armed Bandit (MAB) chamada COLEMAN (*Combinatorial Volatile Multi-Armed BANDiT*). Problemas MAB são uma classe de problemas de decisão sequencial que são intensamente estudados para resolver o dilema de EvE. O problema de TCPCI enquadra-se na categoria volátil e combinatorial, pois múltiplos braços (casos de teste) necessitam ser selecionados, e eles são adicionados ou removidos ao longo dos ciclos. COLEMAN foi avaliada em diferentes sistemas do mundo real, orçamentos de teste, funções de recompensa, e políticas MAB, em relação a diferentes abordagens da literatura, e também no contexto de Sistemas Altamente Configuráveis (do inglês *Highly-Configurable Software*, HCS). Diferentes indicadores de qualidade foram utilizados, englobando diferentes perspectivas tais como a eficácia da detecção de defeitos (com e sem considerar custo), rápida detecção de defeitos, redução do tempo de teste, tempo de priorização, e acurácia. Os resultados mostram que a abordagem COLEMAN é promissora e endossam sua aplicabilidade no problema de TCPCI. Em comparação com RETECS, uma abordagem do estado da arte baseada em Aprendizado por Reforço, COLEMAN apresenta uma melhor eficácia em detectar defeitos em $\approx 82\%$ dos casos, e detecta-os mais rapidamente em 100% dos casos. COLEMAN gasta um tempo negligível, menos do que um segundo para executar, e é mais estável do que a abordagem RETECS, ou seja, melhor se adapta para lidar com os picos de defeitos. Quando comparada com uma abordagem baseada em busca, COLEMAN provê soluções próximas das ótimas em $\approx 90\%$ dos casos, e soluções razoáveis em $\approx 92\%$ dos casos em comparação com uma abordagem determinística. Portanto, a contribuição deste trabalho é introduzir uma abordagem eficiente e eficaz para o problema de TCPCI.

Palavras-chave: Teste de Software, Integração Contínua, Priorização de Casos de Teste, Multi-Armed Bandit.

ABSTRACT

Continuous Integration (CI) is a practice commonly and widely adopted in the industry to allow frequent integration of software changes, making software evolution faster and cost-effective. In CI environments, Regression Testing (RT) is fundamental to ensure that changes have not adversely affected existing features of the system. However, RT is an expensive task. To reduce RT costs, the use of Test Case Prioritization (TCP) techniques plays an important role. These techniques attempt to identify the test case order that maximizes specific goals, such as early fault detection. Recently, many studies on TCP in CI environments (TCPCI) have arisen, but few pieces of work consider CI particularities, such as the time constraint and the test case volatility, that is, they do not consider the dynamic environment of the software life-cycle in which new test cases can be added or removed (discontinued) over time. The test case volatility is a characteristic related to the Exploration versus Exploitation (EvE) dilemma. To solve such a dilemma an approach needs to balance: i) the diversity of the test suite; and ii) the quantity of new test cases and test cases that are error-prone or that comprise high fault-detection capabilities. To deal with this, most approaches use, besides the failure-history, other measures that rely on code instrumentation or require additional information, such as testing coverage. However, maintaining this information updated can be difficult and time-consuming, not scalable due to the test budget of CI environments. In this context, and to properly deal with the TCPCI problem, this work presents an approach based on Multi-Armed Bandit (MAB) called COLEMAN (*Combinatorial VOLatiLE Multi-Armed BANDiT*). The MAB problems are a class of sequential decision problems that are intensively studied for solving the EvE dilemma. The TCPCI problem falls into the category of volatile and combinatorial MAB, because multiple arms (test cases) need to be selected, and they are added or removed over the cycles. COLEMAN was evaluated under different real-world software systems, time budgets, reward functions, and MAB policies, against different approaches from the literature, and also considering the Highly-Configurable Software context. Different quality indicators were used to encompass different perspectives such as fault detection effectiveness (and with cost consideration), early fault detection, test time reduction, prioritization time, and accuracy. The outcomes show that COLEMAN is promising and endorse its applicability for the TCPCI problem. COLEMAN outperforms RETECS, a state-of-the-art approach based on Reinforcement Learning, and stands out mainly regarding fault detection effectiveness (in $\approx 82\%$ of the cases) and early fault detection (in 100%). COLEMAN spends a negligible time, less than one second to execute, and is more stable than RETECS, that is, adapts better to deal with peak of faults. When compared with a search-based approach, COLEMAN provides near-optimal solutions in $\approx 90\%$ of the cases, and in comparison with a deterministic approach, provides reasonable solutions in 92% of the cases. Thus, the main contribution of this work is to provide an efficient and efficacious MAB-based approach for the TCPCI problem.

Keywords: Software Testing, Continuous Integration, Test Case Prioritization, Multi-Armed Bandit.

LIST OF FIGURES

2.1	First-in, First-out procedure in the <i>SW</i> (adapted from Li et al. [50]).	25
2.2	Difference between Standard <i>MAB</i> and Volatile <i>MAB</i> (adapted from Bnaya et al. [7]).	26
2.3	Overview of Continuous Integration, Delivery, and Deployment (extracted from Prado Lima and Vergilio [75]).	28
3.1	PRISMA flow diagram.	32
4.1	Overview of the proposed approach and how it is integrated in the test phase of a <i>CI</i> environment.	38
4.2	Heatmap concerning the test cases vs. faults along <i>CI</i> cycles from the System Example.	40
4.3	COLEMAN illustrative example - Commit 1.	41
4.4	COLEMAN illustrative example - Commit 2.	41
4.5	COLEMAN illustrative example - Commit 3.	42
4.6	COLEMAN illustrative example - Commit 4.	42
4.7	COLEMAN illustrative example - Commit 5.	43
4.8	Steps conducted for extracting and modeling build history from Travis CI.	46
4.9	Information extracted from Deeplearning4j system behavior.	47
4.10	Reward Based on Failures (<i>RNFail</i>) function.	51
4.11	Time-Ranked Reward (<i>TimeRank</i>) function.	51
4.12	Accumulative Reward values over the <i>CI</i> cycles from Druid system.	51

LIST OF TABLES

3.1	Inclusion and exclusion criteria applied on systematic mapping..	31
3.2	Number of papers returned by each source.	33
4.1	Functions used by the Reward Functions..	39
4.2	Test Case Set Information.	45
4.3	NAPFD and <i>APFD_c</i> values (mean and standard deviation) for <i>MAB</i> policies using time budget of 50%..	49
4.4	Mean and standard deviation Prioritization Time (in seconds) with time budget of 50%..	50
4.5	Mean and standard deviation <i>NAPFD</i> , <i>APFD_c</i> , and <i>NTR</i> values: <i>COLEMAN</i> against <i>RETECS</i>	51
4.6	Mean and standard deviation <i>RFTC</i> values: <i>COLEMAN</i> against <i>RETECS</i>	52
5.1	Summary of the main characteristics of the results obtained in the other evaluations.	55

LIST OF ACRONYMS

ANN	<i>Artificial Neural Network</i>
APFD	<i>Average Percentage of Faults Detected</i>
APFD_c	<i>Average Percentage of Faults Detected with cost consideration</i>
CI	<i>Continuous Integration</i>
CD	<i>Continuous Deployment</i>
CDE	<i>Continuous DELivery</i>
CMAB	<i>Contextual Multi-Armed Bandit</i>
COLEMAN	<i>Combinatorial vOlatiLE Multi-Armed BANDiT</i>
EvE	<i>Exploitation versus Exploration</i>
FIR	<i>Fitness Improvement Rate</i>
FRRMAB	<i>Fitness-Rate-Rank based on Multi-Armed Bandit</i>
FCS	<i>Fuzz Configuration Scheduling</i>
GA	<i>Genetic Algorithm</i>
HCS	<i>Highly-Configurable Software</i>
HITO	<i>Hyper-heuristic for the Integration and Test Order Problem</i>
ITO	<i>Integration and Test Order</i>
LSTM	<i>Long Short Term Memory</i>
MAB	<i>Multi-Armed Bandit</i>
MOEA	<i>Multi-Objective Evolutionary Algorithm</i>
NAPFD	<i>Normalized Average Percentage of Faults Detected</i>
NTR	<i>Normalized Time Reduction</i>
PRISMA	<i>Preferred Reporting Items for Systematic reviews and Meta-Analyses</i>
RETECS	<i>Reinforced Test Case Selection</i>

<i>RFTC</i>	<i>Rank of the Failing Test Cases</i>
<i>RMSE</i>	<i>Root-Mean-Square Error</i>
<i>RNFail</i>	<i>Reward Based on Failures</i>
<i>RL</i>	<i>Reinforcement Learning</i>
<i>RT</i>	<i>Regression Testing</i>
<i>SW</i>	<i>Sliding Window</i>
<i>SUT</i>	<i>System Under Test</i>
<i>TCP</i>	<i>Test Case Prioritization</i>
<i>TCPCI</i>	<i>Test Case Prioritization in Continuous Integration environments</i>
<i>TimeRank</i>	<i>Time-Ranked Reward</i>
<i>TSM</i>	<i>Test Suite Minimization</i>
<i>TCS</i>	<i>Test Case Selection</i>
<i>UCB</i>	<i>Upper Confidence Bound</i>
<i>VMAB</i>	<i>Volatile-multi-Arm bandit</i>
<i>VTs</i>	<i>Variant Test Set Strategy</i>
<i>WTS</i>	<i>Whole Test Set Strategy</i>

PUBLISHED WORK

The following works were published in the last 5 years within the subject of test prioritization and continuous integration.

Journal Papers

- Jackson A. Prado Lima and Silvia R. Vergilio. Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, 121:106268, 2020 [75].
- Jackson A. Prado Lima and Silvia R. Vergilio. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, page 12, 2020 [71].
- Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. Cost-effective learning-based strategies for test case prioritization in Continuous Integration of Highly-Configurable Software. *Empirical Software Engineering*, 2021. Accepted [65].
- Jackson A. Prado Lima and Silvia R. Vergilio. An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration. *Journal of Software Engineering Research and Development*, 2021. Submitted [77].

Conference Papers

- Jackson A. Prado Lima and Silvia R. Vergilio. Multi-armed bandit test case prioritization in continuous integration environments: A trade-off analysis. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing, SAST'20*. ACM, 2020 [72].
- Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–11, 2020 [64].
- Jackson A. Prado Lima and Silvia R. Vergilio. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration environments. *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Aug 2021. Journal First Track. Presentation [76].

- Enrique A. Roza, Jackson A. Prado Lima, Silvia R. Vergilio, and Rogério C. Silva. Machine Learning Regression Techniques for Test Case Prioritization in Continuous Integration Environment. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021. Approved [85].

Papers not included in the thesis

Besides that, I have been involved with the following works, which are not included in the thesis, as they focus on topics that are not related with the main context of the PhD project.

Journal Papers

- Jackson A. Prado Lima and Silvia R. Vergilio. A systematic mapping study on higher order mutation testing. *Journal of Systems and Software*, 154:92–109, 2019 [69].
- Henrique N. Silva, Jackson A. Prado Lima, Silvia R. Vergilio, and Andre T. Endo. A Mapping Study on Mutation Testing for Mobile Applications. *Software Testing, Verification and Reliability*, page 23, 2021 [89].

Conference Papers

- Jackson A. Prado Lima and Silvia R. Vergilio. A Multi-Objective Optimization Approach for Selection of Second Order Mutant Generation Strategies. In *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing, SAST*. ACM, 2017 [68].
- Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines. In *Proceedings of the 31st Brazilian Symposium on Software Engineering, SBES'17*, page 114–123, New York, NY, USA, 2017. ACM [28].
- Jackson A. Prado Lima and Silvia Regina Vergilio. Search-Based Higher Order Mutation Testing: A Mapping Study. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing, SAST'18*, page 87–96, New York, NY, USA, 2018. Association for Computing Machinery [80].
- Jackson A. Prado Lima and Silvia Regina Vergilio. Comparing low level heuristics selection methods in a higher-order mutation testing approach. In *Proceedings of the IX Workshop on Search Based Software Engineering, WESB*, 2018 [79].

CONTENTS

1	INTRODUCTION	18
1.1	MOTIVATIONS	19
1.2	OBJECTIVES.	20
1.3	TEXT ORGANIZATION.	21
2	BACKGROUND	23
2.1	MULTI-ARMED BANDIT	23
2.2	VOLATILE AND COMBINATORIAL MAB/CMAB.	25
2.3	TEST CASE PRIORITIZATION.	26
2.4	CONTINUOUS INTEGRATION ENVIRONMENTS.	28
2.5	CONCLUDING REMARKS	29
3	RELATED WORK	30
3.1	MULTI-ARMED BANDIT FOR SOFTWARE TESTING.	30
3.2	TEST CASE PRIORITIZATION IN CONTINUOUS INTEGRATION ENVI- RONMENTS	31
3.3	CONCLUDING REMARKS	36
4	PROPOSED APPROACH.	37
4.1	OVERVIEW.	37
4.2	MAB POLICIES	37
4.3	CREDIT ASSIGNMENT.	38
4.3.1	Reward Functions	39
4.4	COLEMAN'S ILLUSTRATIVE EXAMPLE	40
4.5	EVALUATION DESCRIPTION	43
4.5.1	Quality Indicators	44
4.5.2	Systems Under Test	45
4.5.3	Data Collection	46
4.5.4	Parameters Setting	47
4.5.5	Threats to Validity	48
4.6	RESULTS AND ANALYSES	48
4.6.1	RQ1: <i>COLEMAN</i> Configuration	48
4.6.2	RQ2: <i>COLEMAN</i> Applicability	50
4.6.3	RQ3: Comparing <i>COLEMAN</i> and <i>RETECS</i>	52
4.7	CONCLUDING REMARKS	54
5	OTHER EVALUATIONS	55
5.1	COMPARISON WITH A SEARCH-BASED APPROACH	55

5.2	COMPARISON WITH A DETERMINISTIC APPROACH	56
5.3	APPLICATION FOR HCS	57
6	CONCLUSION	58
6.1	LIMITATIONS	59
6.2	CONTRIBUTIONS	59
6.3	FUTURE WORK	60
	REFERENCES	62
	APPENDIX A – TCPCI: A SYSTEMATIC MAPPING STUDY.	71
A.1	INTRODUCTION	71
A.2	BACKGROUND AND RELATED WORK	72
A.2.1	Test Case Prioritization	72
A.2.2	Continuous Integration Environments	72
A.2.3	Related Work	73
A.3	MAPPING PROCESS	73
A.3.1	Definition of Research Questions	73
A.3.2	Definition of the Search String	74
A.3.3	Selection Criteria	75
A.3.4	Conducting the Review	75
A.3.5	Classification scheme, data extraction and dissemination	75
A.4	OUTCOMES	76
A.4.1	Basic Information of the Field	76
A.4.2	Characteristics of the approaches	79
A.4.3	Evaluation Aspects	82
A.5	TRENDS AND RESEARCH OPPORTUNITIES	84
A.5.1	Evolution of the field	84
A.5.2	Type of approaches	84
A.5.3	Application contexts	84
A.5.4	Evaluation of the approaches	85
A.6	THREATS TO VALIDITY	85
A.7	CONCLUDING REMARKS	85
A.8	PRIMARY STUDIES	86
A.9	REFERENCES	87
	APPENDIX B – A MULTI-ARMED BANDIT APPROACH FOR TCPCI .	89
B.1	INTRODUCTION	89
B.2	BACKGROUND	90
B.2.1	Multi-Armed Bandit	90
B.2.2	Continuous Integration Environments	91

B.3	RELATED WORK	91
B.4	PROPOSED APPROACH	92
B.4.1	MAB Policies	93
B.4.2	Credit Assignment	93
B.5	EVALUATION DESCRIPTION	94
B.5.1	Quality Indicators	94
B.5.2	Systems Under Test	95
B.5.3	Parameters Setting	95
B.5.4	Threats to Validity	95
B.6	RESULTS AND ANALYSIS	95
B.6.1	RQ1: COLEMAN Configuration	95
B.6.2	RQ2: COLEMAN Applicability	97
B.6.3	RQ3: Comparing COLEMAN and RETECS.	97
B.7	CONCLUDING REMARKS	99
B.8	REFERENCES	99
	APPENDIX C – MULTI-ARMED BANDIT TCPCI: A TRADE-OFF ANALYSIS	101
C.1	INTRODUCTION	101
C.2	EVALUATED APPROACH	102
C.2.1	Reward Functions	102
C.2.2	FRRMAB	103
C.3	FINDING NEAR-OPTIMAL SOLUTIONS	104
C.3.1	Population Representation	104
C.3.2	Fitness Function.	104
C.3.3	Algorithm	104
C.3.4	Implementation Aspects	104
C.4	EXPERIMENTAL SETUP	105
C.4.1	Evaluation Measures	105
C.4.2	Statistical Analysis	105
C.4.3	Target Systems	105
C.4.4	Execution Parameters	105
C.4.5	Threats to Validity	105
C.5	RESULTS	106
C.5.1	Discussion.	109
C.6	RELATED WORK	109
C.7	CONCLUDING REMARKS	109
C.8	REFERENCES	110

	APPENDIX D – AN EVALUATION OF RANKING-TO-LEARN APPROACHES FOR TCPCI.	111
D.1	INTRODUCTION	111
D.2	BACKGROUND AND RELATED WORK	112
D.2.1	TCP in CI environments	112
D.3	LEARNING-BASED APPROACHES	113
D.3.1	RETECS	113
D.3.2	COLEMAN	114
D.3.3	Reward Functions	114
D.4	EVALUATION METHODOLOGY	115
D.4.1	Evaluation Measures	115
D.4.2	Target Systems	116
D.4.3	Generating Optimal Solutions	116
D.4.4	Executing Learning-based approaches	116
D.5	RESULTS AND ANALYSIS	116
D.5.1	Fault Detection Effectiveness	117
D.5.2	Fault Detection Effectiveness with Cost Consideration.	119
D.5.3	Early Fault Detection and Test Time Reduction	120
D.5.4	Prioritization Time	122
D.5.5	Accuracy	122
D.5.6	Answering our RQ	124
D.5.7	Discussions and Implications	127
D.6	THREATS TO VALIDITY	127
D.7	CONCLUDING REMARKS	128
D.8	REFERENCES	129
	APPENDIX E – LEARNING-BASED PRIORITIZATION OF TEST CASES IN CI OF HCS	130
E.1	INTRODUCTION	130
E.2	MOTIVATION EXAMPLE.	131
E.3	COLEMAN	131
E.4	APPLICATION STRATEGIES.	132
E.5	EVALUATION SETUP.	133
E.5.1	Research Questions	133
E.5.2	Subject System	133
E.5.3	Quality Indicators	133
E.5.4	Applying learning and random approaches	134
E.5.5	Statistical Analysis	135

E.6	RESULTS	135
E.6.1	RQ1: COLEMAN vs Random	135
E.6.2	RQ2: Strategies Applicability	135
E.6.3	RQ3: Comparing VTS and WTS strategies	135
E.6.4	Threats to Validity	138
E.7	RELATED WORK	138
E.8	CONCLUDING REMARKS	139
E.9	REFERENCES	139
	APPENDIX F – COST-EFFECTIVE LEARNING-BASED STRATEGIES FOR TCPCI OF HCS	141
F.1	INTRODUCTION	141
F.2	MOTIVATION EXAMPLE.	142
F.3	TEST CASE PRIORITIZATION IN CI	143
F.3.1	Adopted Approach	143
F.4	PROPOSED STRATEGIES	145
F.5	EVALUATION SETUP.	145
F.5.1	Subject Systems.	146
F.5.2	Quality Indicators	147
F.5.3	Applying Approaches.	147
F.5.4	Statistical Analysis	148
F.6	RESULTS AND ANALYSIS	149
F.6.1	RQ1: Performance of the strategies using COLEMAN and random approach. . .	149
F.6.2	RQ2: Performance of the strategies using COLEMAN and RETECS	151
F.6.3	RQ3: How far the solutions found by all approaches are from optimal solutions .	152
F.6.4	RQ4: Strategies Applicability	154
F.6.5	RQ5: Comparing VTS and WTS strategies	155
F.6.6	Implications and Limitations	158
F.6.7	Threats to Validity	160
F.7	RELATED WORK	160
F.8	CONCLUDING REMARKS	162
F.9	REFERENCES	162

1 INTRODUCTION

With the adoption of the agile paradigm by most software organizations, we observe a growing interest in *Continuous Integration (CI)* environments. Such environments allow more frequently integration of software changes, making software evolution faster and cost-effective [103]. *CI* environments automatically support tasks like build process, test execution, and test results report, allowing software engineers to merge code that is under development or maintenance with the mainline codebase at frequent time intervals [20]. The results are used to resolve problems and locate faults, and a rapid feedback is fundamental to reduce development costs [43].

During the software life-cycle, there are continuous changes, either in the system itself or its environment. The numerous changes made can make the software more complex and different from the original design, decreasing the software quality. After the changes, the software engineers perform *Regression Testing (RT)* to confirm that changes have not adversely affected existing features of the system. However, *RT* is considered one of the most expensive tasks in software maintenance activities [102].

Companies like Google [60], Facebook [86], and Microsoft [24] have adopted *CI*, as well as open-source projects [40] using available *CI* frameworks (i.e., Travis CI and Jenkins). A study shows that every day at Google, an amount of 800K builds, and 150 Million test runs are performed on more than 13K code projects [60]. This amount of builds and testing can require non-trivial amounts of time and resources [15, 30, 40]. Within an integration cycle, *RT* is an activity that takes a significant amount of time. A test set, many times, includes thousands of test cases that take several hours or days to execute [37]. Even though massive parallelism was reported, Google developers must wait 45 minutes to 9 hours to receive testing results [60].

In the scenario aforementioned, to re-execute all test cases is unfeasible. Then it is fundamental to perform *RT* activities in a very cost-effective way. In the literature, there are many techniques to perform *Regression Testing* [102]. Test Case Minimization techniques usually remove redundant test cases, minimizing the test set according to some criterion. Test Case Selection selects a subset of test cases, the most important ones to test the software. *Test Case Prioritization (TCP)* attempts to re-order a test suite to identify an “ideal” order of test cases that maximizes specific goals, such as early fault detection.

In a company, multiple projects may share the same *CI* workflow and *RT* usually runs a time restricted to a specific duration, the test budget. This makes difficult the use of traditional *RT* techniques that usually rely on costly code analysis and instrumentation, what can be time-consuming and produce results that quickly become inaccurate due to the frequent changes [23]. *TCP* techniques are more suitable in the presence of time constraints, and it has advantages concerning other techniques because *TCP* considers all the test cases, decreasing the risk of reducing code coverage by discarding some of them [19]. Such a characteristic is attractive in the *RT* scenario, in which there are limited resources, thus, it may not be possible to execute the entire *RT* suite [84, 102]. Moreover, ideally, the test set should be executed to maximize early fault detection. However, fault-detection information is unknown until the testing is finished [102]. *TCP* techniques based on failure history can be used to overcome such a difficulty [37].

Although we observe advantages in the use of *TCP* techniques, most of them require adaptations to deal with the *CI* particularities, such as test case volatility, testing budget, and limited resources. Test case volatility is related to the dynamic environment of the software life-cycle in which new test cases can be added or removed (discontinued). The speed up of the

TCP approaches is an important factor to be considered due to the high frequency of changes in *CI* environments and the high *RT* cost. Approaches that require exhaustive analysis are costly and inefficient [23] because the time available to run the prioritized test suite can be reduced if prioritization takes too long [39]. For this reason, the application of search-based techniques or other ones that require extensive code analysis and coverage may be unfeasible, due to the time constraints and the test budget.

To deal with the *Test Case Prioritization in Continuous Integration environments (TCPCI)*, some approaches have recently appeared in the literature [13, 16, 37, 55, 56, 58, 90]. The great majority are not adaptive, that is, they do not learn with past prioritizations and they do not consider test case volatility. Learning approaches based on historical failure data have been proposed to overcome some of these limitations.

Ranking-to-learn approaches learn from past prioritization, based on the rewards obtained from the feedback of previously used ranks. The main idea is to maximize the rewards [71, 90]. These approaches are more robust regarding the volatility of the test cases, code changes, and the number of failing tests [6]. However, the challenge is to search for early fault detection in the failure-history of past test cases, but also to explore new test cases. This is related to the *Exploitation versus Exploration (EvE)* dilemma [47], and is a consequence of the test budget, since whether only error-prone test cases are considered without diversity, some test cases can never be executed.

The approach called *RETECS* [90] deals with the *EvE* dilemma and considers the test budget by using historical test data and *Reinforcement Learning (RL)* [92]. *RETECS* reached the best performance using an *Artificial Neural Network (ANN)*, which is capable of handling a large amount of decisions/states. However, determining why an *ANN* makes a particular decision is a hard task (black box) [5]. Although many studies have arisen recently in the *TCPCI* context. The existing *TCPCI* techniques have limitations, mainly regarding the *CI* particularities. In this context, there is space for innovation [75].

1.1 MOTIVATIONS

The *TCPCI* problem requires approaches that consider the dynamic characteristics of the *CI* environments, the test case volatility, and the test budget. Moreover, such approaches should properly deal with the *EvE* dilemma, that is, an approach needs to balance: i) the diversity of test suite, and ii) the quantity of new test cases and test cases that are error-prone or that comprise high fault-detection capabilities. *Multi-Armed Bandit (MAB)* [2, 82] is a technique intensively studied for solving the *EvE* dilemma. In probability theory, *MAB* problems are a class of sequential decision problems that have many similarities to *RL*.

MAB is considered to be a “lite” form (one-state) of *RL* but *MAB* presents some advantages. *MAB* does not require context information and its actions affect only the immediate reward [104]. In contrast, *RL* actions change the state and it needs to handle the state space, as well as to rely on function approximation to evaluate the value of being in a particular state and taking a specific action.

Traditionally, *MAB* consists of slot machines with K arms, each giving a reward of an unknown probability distribution. The player has a leverage number, and his/her purpose is to choose which leverage lever gives him the best reward. In order to select one of the several arms, a decision-maker (policy) is used. In our work, we conjecture that the use of *MAB* to solve the *TCPCI* contributes to enhance *TCP* in *CI* environments. Modeling a *MAB*-based approach for *TCPCI* problem gives us some advantages in relation to studies found in the literature, as follows:

- It learns how to incorporate the feedback from the application of the test cases thus incorporating diversity in the test suite prioritization;
- It uses a policy to deal with the *EvE* dilemma, thus mitigating the problem of beginning without knowledge (learning) and adapting to changes in the execution environment, for instance, the fact that some test cases are added (new test cases) and removed (obsolete test cases) from one cycle to another (volatility of test cases);
- It is model-free. To consider the dynamic characteristics of the *CI* environments, there are no initial concept of the environment and the approach primarily rely on learning.
- The technique is independent of the development environment and programming language, and does not require any analysis in the code level;
- It is more lightweight, that is, needs only the historical failure data to execute, and has higher performance.

As far as we are aware, no work in the literature evaluates *MAB* approaches in *TCPCI* context, as well as studies in probabilistic theories. Furthermore, no work in the literature considers the use of standard *MAB* with the characteristics of combinatorial and volatile *MAB*, even considering other software engineering fields.

1.2 OBJECTIVES

This work aims to investigate the advantages of *MAB*-based approach in the *TCPCI* context. The hypothesis of this work is that a *MAB*-based approach can prioritize test cases in the *CI* context in a very cost-effective way, such as prioritizing the tests quickly, providing early fault detection, reducing the time spent running a test set, and outperforming approaches from the literature.

To test this hypothesis, we propose *COLEMAN*, a *MAB*-based approach which formulates the *TCPCI* problem as a *MAB* problem and considers characteristics of volatile and combinatorial *MAB*. In this context, each arm is a test case, and the reward is the feedback obtained when the prioritized test suite is applied. However, the standard *MAB* does not consider the dynamic environment (volatility) where the arms can be added (new test cases) or removed (discontinued/removed test cases), and its probabilities are not fixed. Besides that, in the *TCPCI* context, the policy needs to select multiple arms (test cases). This kind of *MAB* problem falls into the category of volatile and combinatorial *MAB*.

We designed *COLEMAN* to be generic regarding the programming language in the system under test, and adaptive to different contexts and testers' guidelines. Hence, we aim to evaluate the applicability in *CI* environments, that is, if we can use *COLEMAN* in practice. For this end, we conducted a broad number of studies following a rigorous methodology. The studies conducted are summarized as follows:

- We evaluated our approach against *RETECS* [90], a *RL* approach, and considered the state-of-the-art in *TCPCI*. For this, we used 11 real-world software systems (chosen by popularity), five metrics, five *MAB* policies (one of them is a random strategy), three time budgets, and two reward functions.
- We evaluated our approach against a *Genetic Algorithm* (*GA*). We provide a trade-off analysis about how far we are from optimal solutions, and we discuss the use of the search-based approach in the *TCPCI* context and its associated costs. We used seven real-world software systems, three metrics, three time budgets, and two reward functions.

- We evaluated our approach and a learning approach against a Deterministic approach. We compare the solutions regarding six measures, 12 real-world software systems, three time budgets, and two reward functions. Our findings have some implications for: i) application of the approaches: we present guidelines that include the choice of the reward function, cost consideration regarding test case duration, and characteristics of the systems and budgets; ii) identification of limitations and possible improvements: we analyze some aspects regarding the number of test cases, failures distribution over test cases and *CI* cycles. Such aspects are drawbacks for the learning approaches and constitute gaps for future research; and iii) benchmark construction: we identify complex prioritization cases.
- We evaluated our approach in the *Highly-Configurable Software (HCS)* context. We proposed strategies to be used by learning approaches to deal with *HCS*: *Variant Test Set Strategy (VTS)* and *Whole Test Set Strategy (WTS)*. In summary, we considered two real-world software systems, five metrics, and three time budgets. We compared *COLEMAN* against *RETECS* and Random approaches.

In this way, this work contributes to the *TCP* field proposing a *MAB*-based approach for *CI* environments that focus on the *CI* particularities, such as test case volatility. Such an approach is model-free and lightweight, capable of automating the *TCP* in *CI* environments while learning with previous prioritization. We performed an in-depth investigation of *MAB* in *TCPCI* considering real-world software systems. The results provide evidence to support the claim that our approach can provide reasonable and near-optimal solutions, as well as our approach outperforms an approach from the state-of-the-art.

1.3 TEXT ORGANIZATION

This work is organized as follows:

- **Chapter 2 - Background:** This chapter describes the background for the understanding of this work. It presents concepts related to Multi-Armed Bandit, Test Case Prioritization, and Continuous Integration environments.
- **Chapter 3 - Related Work:** This chapter presents and describes related work. In this chapter, we present papers that apply *MAB* in software testing and the work on *TCPCI* context.
- **Chapter 4 - Proposed Approach:** This chapter introduces *COLEMAN*, a *MAB*-based approach, by describing the *MAB* policies used, adaption necessary to encompass the combinatorial and volatile characteristics, the reward functions to assign individual reward for each test case, and results from the approach evaluation.
- **Chapter 5 - Other Evaluations:** This chapter presents a summary of other evaluations conducted to assess the feasibility of *COLEMAN*.
- **Chapter 6 - Conclusion:** This chapter presents a summary of the thesis, limitations, and future work.
- **Appendix A to F:** The appendices present the full text of the articles published on the subject of this thesis.

- **Appendix A:** “*Test Case Prioritization in Continuous Integration Environments: A Systematic Mapping Study*”. This article is a systematic mapping on *TCPCI*, and it allowed us to observe the research gaps and main characteristics needed in the approaches to deal with *TCPCI*.
- **Appendix B:** “*A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments*”. Based on the systematic mapping performed (Appendix A) and the research gaps identified, we proposed *COLEMAN*, a *MAB*-based approach, in order to mitigate the main *CI* particularities with a negligible prioritization time. We compare our approach against to *RETECS* (considered as state-of-the-art in *TCPCI*).
- **Appendix C:** “*Multi-Armed Bandit Test Case Prioritization in Continuous Integration Environments: A Trade-off Analysis*”. Due to the promising results from the paper presented in Appendix B, we started an in-depth *COLEMAN*’s evaluation. In this paper, we compare *COLEMAN* against a *GA* algorithm, and we provide a trade-off analysis regarding near-optimal solutions.
- **Appendix D:** “*An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration*”. In contrast with the paper presented in Appendix C, we compare the solutions produced by *COLEMAN* and *RETECS*, with a deterministic approach. This evaluation aims to observe how far the solutions produced by the proposed approaches in the literature are from optimal solutions.
- **Appendix E:** “*Learning-based Prioritization of Test Cases in Continuous Integration of Highly-Configurable Software*”. We evaluate *COLEMAN* in the *HCS* context. Besides that, we proposed two strategies to deal with *HCS* and allow ranking-to-learn algorithms to be easily applied in the this context: *Variant Test Set Strategy* and *Whole Test Set Strategy*.
- **Appendix F:** “*Cost-effective learning-based strategies for test case prioritization in Continuous Integration of Highly-Configurable Software*”. In Appendix E, we observed promising results. Then, we extended the study to encompass a more extensive system, more measures, and provide an in-depth analysis.

2 BACKGROUND

In this chapter, we review background related to our work: Multi-Armed Bandit, Combinatorial and Volatile Multi-Armed Bandit; Test Case Prioritization; and Continuous Integration environments.

2.1 MULTI-ARMED BANDIT

MAB problems [82] are sequential decision problems related to the *EvE* dilemma [47]. This means that, for such problems, solutions with the best performance (exploitation) are desired, but it is also important to ensure diversity (exploration), that is, dissimilar solutions.

The *MAB* problem is related to the scenario in which a player plays on a set K of slot machines (or arms/actions) that even identical produce different gains. After a player pulls one of the arms a_i , $\forall i \in K$, in a turn t , a reward $(q_{i,t})$ is received drawn from some unknown distribution, thus aiming to maximize the sum of the rewards.

A policy γ is a strategy that chooses, at each time t , the next arm to pull based on previously observed rewards and decisions. The *MAB* problem is to determine the policy that maximizes the expected cumulative reward [10] over the *EvE* dilemma. A review of the main *MAB* policies proposed in the literature is presented in [47]. Among them, we can mention the ϵ -greedy policy [95], a policy widely used due to its simplicity. At each time, such a policy evaluates the arms and defines an empirical quality estimate $\hat{q}_{i,t}$ based on previous executions. The $\hat{q}_{i,t}$ value of an arm i in the time t is based on the sum of its previous rewards divided by the number of times that i has been pulled. After, the policy selects, with a probability $1 - \epsilon$, the arm with the highest $\hat{q}_{i,t}$ value (exploitation), or with a probability ϵ , randomly selects an arm (exploration). The parameter ϵ is the key to balance the *EvE* dilemma in the ϵ -greedy policy.

The *MAB* policy called *Upper Confidence Bound (UCB)* provides a smarter way to deal with the *EvE* dilemma and ensures asymptotic optimality in terms of the total cumulative reward [2]. Most of the recent policies are *UCB*-based. In a policy based on *UCB* the i th arm has an empirical quality estimate $\hat{q}_{i,t}$ (the average of the rewards obtained up to the given time instant) and a confidence interval that depends on the number of times, n_i , the arm has been applied before. At each time point t , the selection of the best arm is performed based on the arm with the best upper bound of the confidence interval, according to Equation 2.1:

$$\text{Select } a_t = \underset{i \in K}{\operatorname{argmax}} \left(\hat{q}_{i,t} + \sqrt{\frac{2 \times \ln_{j=1}^K n_{j,t}}{n_{i,t}}} \right) \quad (2.1)$$

where the exploitative first term favors the arms with best empirical rewards, while the exploratory second term favors the infrequently tried arms.

When the rewards are usually among some real-value interval, the *EvE* balance may “break”. To solve this problem, Fialho [27] introduced a scaling factor C in Equation 2.2.

$$\text{Select } a_t = \underset{i \in K}{\operatorname{argmax}} \left(\hat{q}_{i,t} + C \times \sqrt{\frac{2 \times \ln_{j=1}^K n_{j,t}}{n_{i,t}}} \right) \quad (2.2)$$

If exploration is preferable, then C must be increased. On the other hand, if exploitation must be focused, C is decreased. In this thesis the classical *UCB* is named *UCB1*, and the adapted *UCB* proposed by Fialho [27] is named *UCB*.

Another *UCB*-based *MAB* policy is the *Fitness-Rate-Rank based on Multi-Armed Bandit (FRRMAB)*, a state policy that has presented good results in the Adaptive Operator Selection context [50]. In this policy, the best arm is chosen according to Equation 2.3:

$$\text{Select } a_t = \underset{i \in K}{\operatorname{argmax}} \left(FRR_{i,t} + C \times \sqrt{\frac{2 \times \ln \prod_{j=1}^K n_{j,t}}{n_{i,t}}} \right) \quad (2.3)$$

where the goal is, at each time point, to select the best arm from a set of arms which has an empirically estimated value ($FRR_{i,t}$) in a range that depends on the number of times ($n_{i,t}$) that has been applied previously. Similary to *UCB*, the parameter C control the trade-off between exploitation and the exploration.

This policy consists of two procedures: credit assignment and operator (arm/action) selection. *Credit Assignment* (Algorithm 2.1) refers to a reward procedure that takes into account the impact observed in the most recent applications. In the credit assignment, *FRRMAB* policy changed the *UCB* quality estimator (Equation 2.2) by a rank-based method that uses the *Fitness Improvement Rate (FIR)* method. In this procedure, the first step is to calculate the *FIR* for each arm and not use the raw rewards values. The direct use of these values could deteriorate the efficiency of the algorithm [27, 50].

Algorithm 2.1: *Credit Assignment* procedure from *FRRMAB* (adapted from Li et al. [50]).

```

1 begin
2   foreach  $i \in K$  do
3      $Reward_i = 0.0$ ;
4      $n_i = 0$ ;
5   end
6   foreach  $element \in SlidingWindow$  do
7      $i = element.GetArm()$ ;
8      $FIR = element.GetFIR()$ ;
9      $Reward_i = Reward_i + FIR$ ;
10     $n_i ++$ ;
11  end
12  Ranking the rewards in a descending order ( $Reward_i$ );
13   $Rank_i = \text{ranking value of } Reward_i$ ;
14  foreach  $i \in K$  do
15     $Decay_i = D^{Rank_i} \times Reward_i$ ;
16  end
17   $DecaySum = \sum_{i=1}^K Decay_i$ ;
18  foreach  $i \in K$  do
19     $FRR_i = \frac{Decay_i}{DecaySum}$ ;
20  end
21 end

```

After, the *FIR* values are stored in a given *SW* organized as a first-in, first-out queue used to evaluate the W recent applications. In this way, the most recent values are added to the end while the oldest records are removed from the beginning to maintain a constant size. Figure 2.1 shows how to store the *FIR* values related to the respective arm.



Figure 2.1: First-in, First-out procedure in the *SW* (adapted from Li et al. [50]).

Through the use of *SW* it is possible to evaluate an arm without it is hampered by its performance at a very early stage, which may be irrelevant to its current performance. Thus, it is guaranteed that the *FIR* information in *SW* refers to a current search situation [50]. Subsequently, the $Reward_i$ of an arm i is calculated, by the sum of all *FIR* values for the arm i in *SW*. Next, a descending ranking of all rewards from the arm in *SW* is determined. Thus, we define a $Rank_i$ that represents the ranking value of an arm i , which prioritizes the best arms. Then, a decay factor $D \in 0, 1$ is used in order to transform the initial reward according to the relative position about the reward from the other arms ($Decay_i$). Lower values of D means great influence of the best arm. Finally, the decayed values of the rewards are normalized and resulting in the *Fitness-Rate-Rank* (*FRR*) for each arm i . These values (FRR_i) are then used by the arm selection procedure.

The second procedure (Algorithm 2.2), randomly selects an arm until all arms are selected, then uses the *FRRMAB* policy to evaluate each arm and selects the best one. This procedure is similar to the original *UCB* policy [2]. The main difference is the use of *FRR* values with quality indexes instead of the average of all the rewards received by a given operator [50]. Given that n_i indicates the number of times an arm i has been used in its recent W applications in *SW*. Besides that, *FRRMAB* starts acting only when all the arms have been previously used at least once in the search, thus giving all of them chances to be equally selected.

Algorithm 2.2: Operator selection procedure from *FRRMAB* (adapted from Li et al. [50]).

```

1 begin
2   if all arms not been selected yet then
3      $a_t = \text{an arm from set } K \text{ chosen randomly};$ 
4   else
5      $a_t = \underset{i \in K}{\operatorname{argmax}} (FRR_{i,t} + C \times \sqrt{\frac{2 \times \ln \sum_{j=1}^K n_{j,t}}{n_{i,t}}});$ 
6   end
7 end
```

2.2 VOLATILE AND COMBINATORIAL MAB/CMAB

Although *MAB* policies can be used to solve many problems, in many real-world scenarios, a policy needs to select multiple arms in each time. This is the case of our scenario, *TCPCI*

problem, where we need to prioritize a test case set, and we can consider a test case as an arm. This kind of *MAB* problem is categorized as *combinatorial bandit*, where a set of arms are chosen at each time t rather than one individual arm, that is, we are required to pull a fixed number m of arms from a set of arms K , such that $1 \leq m \leq |K|$ [1].

In addition to this and considering the inherently dynamic nature of our problem, the arms available at each time may change dynamically over time. In this sense, this work is based on a *MAB* variant known as *Volatile-multi-Arm bandit* (*VMAB*) [7]. *VMAB* considers that the arms can “appear” or “disappear” unexpectedly in each time. In *VMAB*, each arm $i \in K$ is associated with a *lifespan* given by an interval of time ($start_{t,i}, end_{t,i}$), during which this arm is available. The arm’s *lifespans* are unknown in advance. Figure 2.2 shows the difference between Standard *MAB*, that has a fixed set of K arms, and its variant, *VMAB*. The bold arm (slot machine) represents the optimal arm in time t_j .

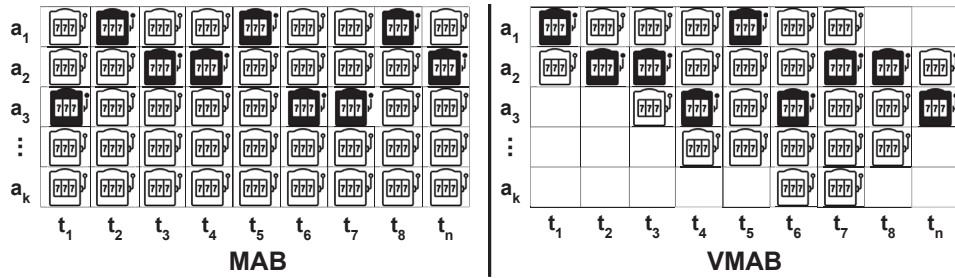


Figure 2.2: Difference between Standard *MAB* and Volatile *MAB* (adapted from Bnaya et al. [7]).

2.3 TEST CASE PRIORITIZATION

Changes are frequent during the software life-cycle. In order to check if any change has not adversely affected existing features of the system, software engineers perform *RT*. The most straightforward *RT* approach, known as re-test all, executes all the current test cases in the test suite. However, such an approach presents high costs [102].

Many techniques have been studied to aid the *RT* process, seeking to reduce the effort required in various ways [102]. Among these techniques, the main branches are *Test Suite Minimization* (*TSM*), *Test Case Selection* (*TCS*), and *Test Case Prioritization*. *TSM* seeks to maintain the most important test cases in the test suite, removing obsolete or redundant test cases. *TCS* aims to select a subset of test cases, the most important ones to test the software. *TCP* attempts to re-order a test suite to identify an “ideal” ordering of test cases that maximizes specific goals, such as early fault detection.

TCP techniques have advantages with respect to other techniques because they consider the whole test suite, consequently decreasing the risk of reducing code coverage by discarding some test cases [19]. On the other hand, this can be, in some cases, time-consuming. However, as *TCP* allows the most crucial test cases are first executed, this characteristic is attractive (and suggested), given that in a *RT* scenario, there are limited resources, that is, it may not be possible to execute the entire *RT* suite [84, 102].

Given a test suite T , the set PT of all possible permutations of T , and a function f that determines the performance of a given prioritization T'' from PT to real numbers, the *TCP* problem aims at finding the best T' to achieve certain specific criteria measured by f [84]. The *TCP* problem can then be formulated as [84]:

$$T' \in PT \text{ s.t. } \forall T'' \in PTT'' \neq T' fT' \geq fT'' \quad (2.4)$$

According to Rothermel et al. [84], some objectives of *TCP* techniques are: (i) to increase the fault detection rate of a test suite already in the beginning of the *RT* execution; (ii) to increase the system code coverage under test; (iii) to increase high-risk fault detection rate; and (iv) to increase the probability to reveal faults related with specific code changes. To achieve such objectives, many *TCP* techniques were proposed in the literature. According to the information used in the prioritization, they can be classified into different categories [102]. They are:

- **Cost-aware:** prioritizes test cases based on the cost of the test cases, because the costs of them cannot be equal;
- **Coverage-based:** prioritizes test cases based on the code coverage;
- **Distribution-based:** prioritizes test cases based on the distribution of the test case profiles;
- **Human-based:** prioritizes test cases based on factors that (human) testers deem the most important;
- **History-based:** prioritizes test cases based on test case execution history information and code changes;
- **Requirement-based:** prioritizes test cases based on information extracted from software requirements;
- **Model-based:** prioritizes test cases based on information extracted from models, for instance, *UML* (*Unified Modeling Language*) models;
- **Probabilistic:** prioritizes test cases based on probabilistic theories;
- **Others:** prioritizes test cases based on other kind of information not included in the other categories, for instance, prioritize the tests cases using search-based algorithms.

To evaluate the effectiveness of *TCP* techniques, several evaluation metrics were proposed. Reviews on *TCP* [14, 44] reported the most frequently used evaluation metrics. Among these metrics, we can mention *Average Percentage of Faults Detected* (*APFD*) [83] and its variations *Average Percentage of Faults Detected with cost consideration* (*APFDc*) [21] and *Normalized Average Percentage of Faults Detected* (*NAPFD*) [81, 90].

APFD indicates how quickly a set of prioritized test cases (T') can detect faults present in the application being tested, and its value is calculated from the weighted average of the percentage of detected faults. Higher *APFD* value indicates that the faults are detected faster using fewer test cases.

APFDc (Equation 2.5)¹ was proposed to deal with an *APFD* limitation concerning the assumption that all faults have equal severity, and the test cases have equal costs. These assumptions are not possible in practice, and therefore, *APFDc* metric takes into account the fault severity and test cost. Furthermore, if both fault severity and test case costs are identical, *APFDc* can be used to compute the *APFD* value.

$$APFDcT'_t = \frac{\sum_{i=1}^m \sum_{j=TF_i}^n c_j - 0.5c_{TF_i}}{\sum_{j=1}^n c_j \times m} \quad (2.5)$$

¹In this work, we assume the same fault severity for all tests, and the test case duration is the cost associated.

where c_i is the cost of a test case T_i , and TF_i is the first test case from T' that reveals fault i .

The *NAPFD* metric (Equation 2.6) is an extension of the *APFD*. In the *NAPFD*, we consider the ratio between detected and detectable faults within T . This metric is adequate to prioritize test cases when not all of them are executed, and faults can be undetected.

$$NAPFDT' = p - \frac{\frac{1}{n} \text{rank} T'_i}{m \times n} + \frac{p}{2n} \quad (2.6)$$

where m denotes the number of faults detected by all test cases; $\text{rank} T'_i$ is the position of T'_i in T' , if T'_i did not reveal a fault we set $T'_i = 0$; n denotes the number of test cases in T' ; and p denotes the number of faults detected by T' divided by m . *NAPFD* is equal to *APFD* metric if all faults are detected.

2.4 CONTINUOUS INTEGRATION ENVIRONMENTS

In the past, developers adopted a practice to work separately for a long time during the development, and they only integrated their changes to the master branch when they completed their work. However, this practice presents some disadvantages. It is time-consuming, adds unnecessary bureaucratic cost to the projects, and accumulates uncorrected errors for long periods. These factors hampered the rapid distribution of updates to customers.

With the advent of the agile development paradigm, *Continuous Software Engineering* practices have become popular and adopted by most organizations, such as *CI*, *Continuous Deployment* (CD), and *Continuous Delivery* (CDE). Such practices play an important role in agile development, allowing frequent integration, reduced integration effort, test of the changed code, lower number of uncorrected errors for long periods, deployment, delivering a product version at any moment, and quick feedback from the customer in a very rapid cycle [20]. Figure 2.3 presents the relationship between these practices.

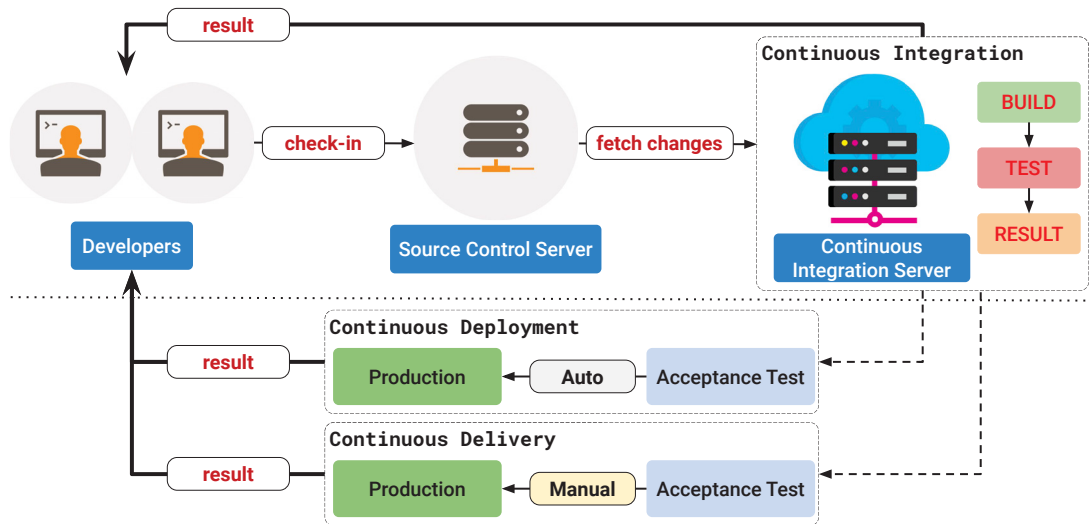


Figure 2.3: Overview of Continuous Integration, Delivery, and Deployment (extracted from Prado Lima and Vergilio [75]).

CI is an essential practice adopted before deployment and delivery. *CI* environments provide automated software building and testing [48], helping to scale up headcount and delivery output of engineering teams, as well as allowing software developers to work independently on features in parallel. When they are ready to merge these features into the end product, they can

do this independently and rapidly. Among the popular open-source *CI* servers we can mention: Buildbot [11], GoCD [31], Integrity [41], Jenkins [42], and Travis CI [93].

CI automated support is very important to *CI* in practice. Zhao et al [103] present some results about the impact of adopting the framework Travis CI on development practices in a collection of GitHub projects. They observed an increase in the number of daily commits (frequency of 78 commits every day) and after some initial adjustments an increase in the number of automated tests.

CDE aims at packing an artifact (the production-ready state from the application) to be delivered for acceptance testing. In this sense, the artifact should be ready to be released to end-users (production) at any given time. On the other hand, *CD* is responsible for automatically packing, launching and distributing the software artifact to production. We can observe that *CI* ensures that the artifact used in the *CD* and *CDE* successfully passed the integration phase.

In order to provide a swift and cost-effective way to validate and launch new software updates, improving fault detection and software quality, *RT* is an essential activity in *CI* environments. This is because to enable rapid test feedback in *CI*, test cycles are restricted to a specific (short) duration. We refer to this time duration of each test cycle as a time budget. Time budgets can vary from a cycle to another, and they include time to select relevant tests for running, to run the tests, and to report test results to developers.

In this way, this makes difficult the use of traditional *RT* techniques that rely on costly code analysis and instrumentation. They are time-consuming and produce results that quickly become inaccurate due to the frequent changes [23]. Besides that, traditional *TCP* techniques require some adaptations to be applied in the *CI* context. The techniques need to consider specific particularities of *CI* environments like: parallel execution of test cases and allocation of resources, the volatility of test cases, that is, the test cases can be added and removed in subsequent commits. To address all these particularities, some approaches have been proposed in the literature, being *TCPCI* an emergent research topic [75].

2.5 CONCLUDING REMARKS

This chapter presented the topics related to this work, including the main concepts about *Multi-Armed Bandit*, *Combinatorial and Volatile MAB*, *Test Case Prioritization*, and *Continuous Integration* environments. Concerning *MAB*, the policies that will be used in this work were presented. The main *TCP* advantages were highlighted for a *CI* environment, which contains many restrictions (*TCPCI* problem).

There are many works on *TCP*, subject of surveys and mappings [14, 44, 102]. However, only few pieces of work address *CI* environments. To reach effective prioritization in such environments, we need techniques to reduce the *CI* cycle time, and a key aspect is a short feedback loop from code commits to test execution reports [55]. It is necessary to detect most of the faults within a test budget, because of this, most of the *TCP* approaches for *CI* environments are based on failure history, that is, they assume test cases which previously failed have a high probability of failing again. This assumption has been explored for *TCP* in the presence of constraints by many authors [45] however, without addressing *CI*.

The next chapter introduces works related to the theme of this thesis by showing the use of *MAB* on software testing, and works on the *TCPCI* problem.

3 RELATED WORK

This chapter describes related work and contains three sections. Section 3.1 presents works that use *MAB* to solve software engineering problems, mainly software testing problems. Section 3.2 presents results from a systematic mapping conducted to find studies in the *TCPCI* context and reviews the main studies related to this work. Section 3.3 concludes this chapter and presents the main differences between related work and our proposal .

3.1 MULTI-ARMED BANDIT FOR SOFTWARE TESTING

In the literature, we can find few studies that use *MAB* on software engineering and all of them are related with software testing. These studies addressed *Integration and Test Order (ITO)* problem [33–36], Software Production Line testing [25, 26, 91], *Fuzz Configuration Scheduling (FCS)* [97], and Test Data Generation [18]. Such works do not specifically address *TCP* problem, and most of them are related with hyper-heuristics [25, 26, 33–36, 91]¹.

Guizzo et al [34] proposed a hyper-heuristic named *Hyper-heuristic for the Integration and Test Order Problem (HITO)* for the *ITO* problem in order to reduce *stubby* costs. The *ITO* problem aims to find a sequence of *units*² to be tested that minimizes the *stubby* cost. A *stub* is an emulation of a unit that is later dropped when the *unit* is developed. The *stubby* cost is related to the spent resources for developing *stubs*.

HITO uses two selection functions (Choice Function [53] and *MAB*³) to select the best low-level heuristic (a combination of mutation and crossover operators) in each mating of a *Multi-Objective Evolutionary Algorithm (MOEA)*. To perform the selection, the low-level heuristics are evaluated concerning its recent improvements. *HITO* was implemented using NSGA-II [17] and evaluated in seven systems, and for all systems outperformed the traditional NSGA-II. In another study, the SPEA2 [107] algorithm is also explored [36] and after that a boarder set of objectives were used [35], as well as the MOEA/DD algorithm [49].

Guizzo et al. [35] explored more objectives (4 objectives) to be addressed for *ITO* problem during the optimization rather than two objectives as explored in previous work [34, 36]. When compared, *HITO* outperformed NSGA-II and MOEA/DD [49]. Concerning the selection functions used by the works mentioned above, CF yielded competitive results when compared to *MAB*, except in [33] where *HITO* was evaluated in Google Guava system with same settings from [35] and CF was better than the *MAB* used.

Using a similar concept from *HITO*, Strickler et al. [91] used hyper-heuristic with *FRRMAB* as selection function to derive products for variability test of Feature Models. First, three *MOEAs* were evaluated, NSGA-II, SPEA2, and IBEA [106]. In the experiments, NSGA-II outperformed the other ones. After, NSGA-II was used to be adapted to a hyper-heuristic using *FRRMAB*. The hyper-heuristic outperformed the traditional NSGA-II.

In the same problem, Ferreira et al. [25] used a hyper-heuristic with the MOEA/D algorithm along with three selection methods: UCB, UCB-V, and UCB-Tuned. In the evaluation, MOEA/D with UCB-Tuned outperformed the other ones without statistical difference. In

¹According to Burke et al. [12] hyper-heuristic is a methodology that automates the design and configuration (tuning) of heuristic algorithms to solve computationally hard problems. It can be used to automatically determine which operator should be applied in the optimization problem, at a given moment.

²The smallest part of the software, for instance, classes, methods, and procedures.

³The study uses Sliding Multi-Armed Bandit.

another evaluation, the authors compared MOEA/D-UCB (due to the low cost concerning with the time consuming) against the algorithm MOEA/D-DRA. The hyper-heuristic outperformed the MOEA/D-DRA. In a later work, Ferreira et al. [26] compared different algorithms: NSGA-II, SPEA2, IBEA, and MOEA/D-DRA, and two selection methods: UCB and Random. NSGA-II using UCB outperformed the other ones.

Woo et al. [97] propose a black-box mutational fuzzing model to find bugs in a given program p by running it on a sequence of inputs generated by random mutation to a given seed input s . The goal is to find values for s , which crash p . Given a list of configurations to be tested and a time budget, the authors dubbed this as the *FCS* problem. *FCS* seeks to maximize the number of unique bugs discovered that runs for a duration predefined. Due to this, the authors modeled *FCS* as *MAB* problem for seed selection.

Degott et al. [18] proposed an approach to guide test generation for graphical user interfaces. The problem was modeled as an instance of the *MAB* problem: (i) the arms are a pair which contains a widget and an action type supported by the test generator; (ii) the probabilities are related with each action triggering an app response; and (iii) the reward is 1 (one) if the app's user interface changed after executing the arm, and 0 (zero) otherwise. The implementation was made as a plug-in for DroidMate-2 [9]. In the study, ϵ -Greedy and Thompson Sampling were evaluated and led to an average coverage increase, respectively, of 18% and 24% when compared to a statically gathered crowd-model [8].

In general, it is possible to notice there are several studies on the use of *MAB* with hyper-heuristics to select operators during the mating. Among the presented studies, none of them have explored the use of *MAB* to address the *TCP* problem.

3.2 TEST CASE PRIORITIZATION IN CONTINUOUS INTEGRATION ENVIRONMENTS

We conducted a systematic mapping to find studies in *TCPCI* context. This section summarized the results of the conducted mapping.

The mapping adopted the guidelines proposed by Petersen et al [63], and followed a research plan including research questions, inclusion and exclusion criteria, construction of the search string and selection of known search databases.

At first, we used the main terms regarding this subject and we built the following search string: ("*continuous integration*") *AND* (*prioritization OR prioritisation*) *AND* (*test OR testing*), which was validated using a control group. Besides, a set of inclusion and exclusion criteria was established (Table 3.1). We performed the search in online repositories that were chosen due to their popularity and because they provide many leading software engineering publications.

Table 3.1: Inclusion and exclusion criteria applied on systematic mapping.

Inclusion Criteria	
I1	The paper is related to Software Engineering area;
I2	The paper is related to <i>TCPCI</i> .
Exclusion Criteria	
E1	Out of scope, not related with <i>TCPCI</i> ;
E2	Not available online;
E3	Not in English;
E4	Abstracts, posters, reviews, conference reviews, chapters, thesis, keynotes, doctoral symposiums and patents.

The search started and finished in October, 2019. The mapping was conducted in a set of steps, presented in Figure 3.1 following the *Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA)* statement [61]. The figure presents, for each search engine, the number of studies found and period covered. As we can see, a total of 818 studies was found, including the period from 1979 to 2020.

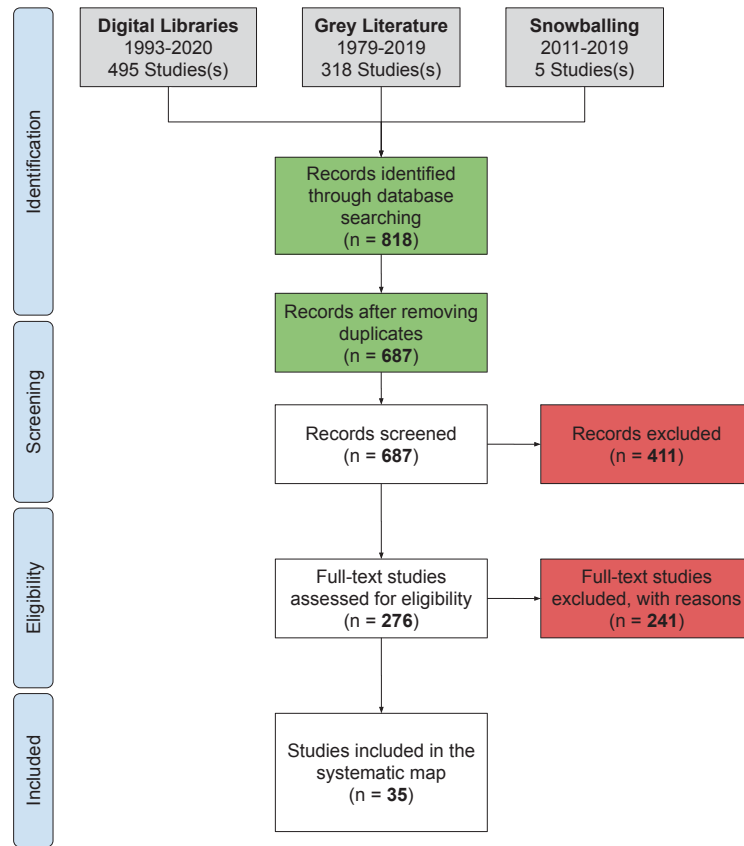


Figure 3.1: PRISMA flow diagram.

For each search engine, the number of studies found is presented, as well as the range of publication data from these studies.

We did not define an initial publication date for the studies, then all the returned papers were included. Table 3.2 presents the data sources used and the period covered, with the data separated by Digital Libraries, Grey Literature, and Snowballing. We performed the mapping in 17 data sources, identifying a total of 818, and covering the period from 1979 to 2020.

We then removed repeated studies, remaining 687. Then, we applied the selection criteria and obtained 276 studies. The selection criteria was applied considering title, abstract, and index terms of the papers (keywords). In case of doubts, reading in the following order was performed: introduction, conclusion, and the entire paper. In the end, 35 papers remain. As a result of this mapping a paper was published at the Information and Software Technology, and the full text is available in Appendix A. In this mapping, we analyzed the details and characteristics of each study found, as well as research gaps and trends. Furthermore, we classified the studies found based on the goal and kind of *TCP* technique used, adopted evaluation measures, and whether they addressed *CI* particularities and testing problems.

We observed a growing interest in this topic. Most of the studies were published in the last four years - considering the year publication of the mapping. Most of the approaches (80%) are based on historical information considering failure and/or test execution history. An

Table 3.2: Number of papers returned by each source.

Source	Studies	Period Covered
Digital Libraries		
ACM	24	2009-2019
EBSCOhost	3	2012-2018
Ei Compendex	31	2009-2019
IEEEExplore	20	2009-2019
ISI Web of Science	27	2009-2019
MIT Libraries	20	2009-2019
ScienceDirect	2	2012-2018
Scopus	35	2009-2019
Springer Link	273	1993-2020
Wiley Online Library	60	2007-2019
Grey Literature		
AWS Whitepapers & Guides	0	-
Google AI	0	-
Google Cloud	2	2018-2019
Google Scholar	129	2001-2019
Microsoft Academic	2	2009-2018
Microsoft Research	0	-
Science.gov	185	1979-2019
Snowballing		
Forward and Backward procedures	5	2011-2019
Total	818	1979-2020

important finding is related to *CI* testing problems and characteristics, for instance, parallel execution and test case volatility, in which few studies address them. Among the studies found in the mapping and in the current literature, the most related studies are those that introduce probabilistic and history-based approaches. These studies are described as follows.

Some approaches have the goal of reducing the amount of resources utilized in a *CI* environment. Liang et al. [51] proposed an approach to prioritize commits based on the test suite failure and execution history. The idea of prioritizing commits can be useful when there are no differences in the fault detection rate between test case sets and there is a line of commits to be executed in the environment.

Other approach that considers multiple test requests was proposed by Zhu et al. [105]. Such an approach uses co-failure distributions of tests, that is, tests that co-fail in previous executions in different sets. The approach considers multiple test requests and they are prioritized based on their failure likelihood and not on their arrival order. The re-prioritization is dynamically performed after each test run and considers that failures are not completely independent.

Elbaum et al. [23] introduced new algorithms for selection and prioritization of test suites, to be used respectively in a pre-submit and pos-submit testing phases. In the first phase, which occurs prior to commit, developers specify modules to be tested and the selection algorithm selects test sets based on failure and execution windows, which allow, respectively, selection based on the fault detection history and of test cases not recently executed. In the post-testing, after commit, the *TCP* algorithm prioritizes test suites considering both windows and a time window. In this way, the algorithm deals with test suite concurrency execution in the pos-submit testing.

Najafi et al. [62] proposed a simple technique to prioritize the test cases similar to Elbaum et al. [23]. In this study, the priority of test cases is given by the ratio between the total of failures found by test cases and the all time spent on executing the tests. In this way, a high priority is given to the test that finds more failures with shorter execution time.

Differently from the previous mentioned works, we assume a sequential execution order of test cases and prioritize test cases in the suite to be executed after the commit, usually considering a test budget. Approaches with such a goal are the most related to ours.

Marijan et al. [56] introduced ROCKET, an approach that, given a test budget, sets a weight for each test case based on the distance of the failure status from its current execution and its execution time. The implementation also uses domain specific heuristics to obtain the prioritized set of test cases and was evaluated in the context of *HCS*. We observe a limitation regarding the execution time. Test cases with an execution time greater than a limit are penalized, and it is possible they are never executed. To set the weight, the prioritization feedback is not considered, nor the total history of failures. An extension is proposed by the authors in [55] to consider other fault detection and different perspectives given a testing budget: business, performance, and technical. The algorithm calculates a weight to each test case considering: failure frequency (business perspective), execution time (performance perspective), and severity and cross-functionality (technical perspective). Such an extension needs additional information related to coverage and features.

In order to analyze the effect of a time window in *TCPCI* context, Marijan et al. [57] compared random ordering, an automatic algorithm for *TCPCI* [56], and manual prioritization. During the analysis, the authors considered a time-limit constraint to run the tests, using different window sizes at each test suite run. According to the authors, changing the window size can impact in the prioritization performance, improving fault detection effectiveness up to a certain limit. Defining an optimal value for the time window, the average percentage of faults detected can increase from 5% to 21%. This study endorses that the use of a time window can improve the prioritization. In this way, we adopt in our work the *FRRMAB* policy which uses a sliding window.

In other analysis, Marijan et al. [59] proposed an approach for reducing long cycle times in DevOps by reducing the duration of test cycles in *CI*. In this study, the authors prioritized the test cases considering historical data information and risk coverage of test suites.

An approach and a tool called TITAN are proposed in [58]. It implements test prioritization and minimization techniques, and provides test traceability and visualization. First, a minimization step adopts the exact method constraint programming to determine a minimum number of test cases (or cost) that cover some requirements that the original test suit covers (regarding *HCS* features). The minimized set is then prioritized in a second step according to previous work [56], by using several criteria fault detection, coverage, execution time and failure impact. Again, additional information, such as feature coverage, is necessary. Moreover, the prioritized set may not contain all available test cases because a minimization step is first conducted.

Xiao et al. [100] proposed a technique for test case prioritization and selection. This technique determines priority of test cases in the same commit, then, the test cases are ordered considering the failure history, test coverage, test size and execution time. We observe that this technique focuses only in test cases which failed recently; consequently, it does not explore new test cases.

Cho et al. [16] proposed an approach, named AFSAC, composed by two stages. First, weights for the test cases are determined using statistical analysis over the failure history. Then

the test cases are reordered using the correlation data of test cases acquired by previous test results.

Haghighatkah et al. [37] presented empirical results that show the use of historical failure knowledge is a strong predictor for *TCPCI*, being effective to catch regression faults earlier without requiring a large amount of historical data. In addition to this, the effectiveness can be improved by using such a knowledge with a diversity measure, calculated by comparing the text of test cases. The idea is not to calculate similarity based on measures that rely on code such as coverage, call methods and so on. After a failure-history based approach the prioritized set is improved with existing test cases that are the most dissimilar.

Azizi [3] proposed an approach based on Information Retrieval that prioritizes test cases based on their textual similarity to the portion of the code that has been changed. However, this technique relies on code analysis.

A problem observed in the approaches is that they do not consider the volatility of the test cases, that is, they do not consider a test case may appear and/or disappear over the cycles. Most of them do not take into consideration the feedback from the prioritization conducted. Some works have addressed this limitation with the use of machine learning.

The approach of Busjaeger and Xie [13] used $SV M^{map}$ to create a model based on five attributes: test coverage of modified code, textual similarity between tests and changes, recent test-failure or fault history, and test age. Such a model is used to predict the fault-proneness of the test cases and prioritize them. However, the attributes need additional information and rely on code instrumentation.

Xiao et al. [99] proposed an approach based on *Long Short Term Memory (LSTM)* Neural Network using historical test data of air-crafts to predict fault-proneness. The authors also suggest a strategy for combining the *LSTM* approach with a deterministic one. Such a combined approach was compared against *ROCKET*, *RETECS* and a random approach. The *LSTM* approach achieved a satisfactory result, and presented better performance when the number of *CI* cycles increases. However, *CI* particularities were not considered when evaluating the *LSTM* performance, such as test budget and prioritization time.

We can see that some works mentioned above introduced learning-based approaches. In this context, Bertolino et al. [6] distinguished and evaluated two kinds of *TCP* learning-based approaches. The first kind, named Learning-to-Rank, uses supervised learning to train a model based on some test features. The model is then used to rank test sets in future commits. The problem with these strategies is that the model may no longer be representative when the commit context changes.

The second kind, named Ranking-to-Learn, is more suitable to the dynamic *CI* context. This strategy learns based on rewards obtained from the feedback of previously used ranks. The main idea is to maximize the rewards. Ranking-to-learn approaches present some advantages. They are more robust regarding the volatility of the test cases, code changes, and the number of failing tests. Next, we describe works on this kind of approach and works that propose reward functions and strategies.

Spieker et al. [90] introduced *RETECS*, an approach to prioritize and select test cases based on Reinforcement Learning which considers as input the test case duration, historical failure data, and previous last execution. The authors compared different variants of Reinforcement Learning agents and the *ANN* variant presented the best results.

Wen et al. [96] adapted *RETECS* to reprioritize the test cases considering associated test cases (relationship between test cases) found through FP-Growth algorithm [38], a mining frequent pattern algorithm. However, this algorithm requires several *CI* cycles to form the associated pattern.

In other studies [87, 88, 98, 101] different reward functions were investigated. The studies provide a picture of the importance of defining an adequate reward function. The reward function has a considerable impact on the approach's performance. For this reason, and to allow a fair comparison against to *RETECS*, the approach most related with our approach, we adopted the same reward functions used by Spieker et al. [90] to evaluate *RETECS*: *RNFail* and *TimeRank*.

To summarize, studies on *TCPCI* context are recent, and the number of them has been increasing. However, few studies consider time constraints and test case volatility in *CI* environments.

3.3 CONCLUDING REMARKS

This chapter presented related work regarding *MAB* in software testing and *TCPCI* context. In one hand, *MAB* is typically used with hyper-heuristics to select operators in optimization algorithms. On the other hand, *TCPCI* techniques tend to use historical test data.

We can summarize some drawbacks identified in related work concerning *TCPCI*. Some of them have different goals from ours, for instance, to reduce server resources considering concurrent test set executions. The most related approaches do not properly deal with the *EvE* problem. This problem regards to the fact that as only a sub-set of the prioritized test cases can be executed regarding its order, some test cases can never be executed given the test budget. To deal with this, most approaches use, besides the failure-history, other measures that rely on code instrumentation or require additional information, such as to calculate code or feature coverage. This can be time-consuming, and to maintain the information updated can be difficult.

The great majority of the studies do not take into consideration the volatility of test cases and feedback from last prioritizations. Differently, our approach considers the test cases volatility and learns with the past prioritizations (*online learning*). It properly deals with the *EvE* dilemma without requiring source code analysis or any initial concept (model) about the system. Few knowledge about each test case is necessary. In this sense, the approach that is most similar to ours is *RETECS* [90]. Differently, our approach uses *MAB*, which allows test cases rewards in a sliding window, and less input information, as well as, context information is not necessary.

In this way, we aim for an approach more lightweight and with higher performance than *RETECS*, as well as that deals appropriately with *EvE* dilemma. Furthermore, *RETECS* has the best performance using an *ANN* [90]. However, neural networks take ample time [108], and determining why an *ANN* makes a particular decision is a challenging task. The next chapter presents the proposed approach.

4 PROPOSED APPROACH

This chapter describes our approach based on Multi-Armed Bandit, namely *COLEMAN* (*Combinatorial VOLatiLE Multi-Armed BANDiT*) for *TCPCI*. This approach was introduced in a paper published in *IEEE Transaction on Software Engineering* (the full text is available in Appendix B), and compared against *RETECS*, an approach based on *RL* that can be considered the state-of-the-art in *TCPCI*.

Next, we present *COLEMAN* and a summary of the results found in the comparison against *RETECS*. Section 4.1 presents an overview of *COLEMAN*. Sections 4.2 and 4.3 describe the adaption necessary in the *MAB* policies and how the historical test data is used by them. Section 4.5 describes the methodology adopted to evaluate our approach, and Section 4.6 presents the results produced in the experiments. Section 4.7 concludes this chapter.

4.1 OVERVIEW

Given the dynamic nature of our problem, our *MAB* approach combines two *MAB* variants: i) combinatorial *MAB*, because we have a set of arms (test cases), and ii) volatile, because at a given time t such set varies, test cases can be added or removed over the software life-cycle. In addition to this, the approach works with a budget (constraint) to execute the test cases prioritized. To ensure diversity of test cases to be executed, it uses *MAB* policies, allowing better exploitation and exploration (*EvE* dilemma).

Figure 4.1 illustrates how our approach works in a *CI* environment. After a successful build, in the test phase, the approach receives as input a set of test cases T_t (arms) available for the current commit (cycle/time) t and, based on the choice order given by a *MAB* policy, generates the prioritized test case set T'_t . In each time t , only one test suite (test case set) is prioritized. Then the system is tested using T'_t and feedback from this test set is collected, containing information, such as the test cases executed in a time limit, the number of failures, the test case failure rank, and so on. This feedback is used by the reward function in the credit assignment procedure to set individual rewards for each test case. In the T'_t evaluation, a fitness value of T'_t is obtained by a quality indicator.¹ This value can be used by testers along with the commits to evaluate the prioritization quality.

Then, the credit assignment procedure calculates the rewards for each arm, test case $t'_c \in T'_t$, which are stored in a historical database to be used in the next commits by the policies. In the end, results are reported back to the *CI* server. The *CI* server sets the result from the cycle and notifies the parties interested in the cycle.

Next, we detail the main elements of *COLEMAN*: *MAB* policies (Section 4.2) and credit assignment (Section 4.3).

4.2 MAB POLICIES

Considering the traditional behavior of a *MAB* policy, the best arm (test case) t_c from a set T_t at each time t is chosen and applied. However, we are working with *TCP* problem where a prioritized test set is used. To this end, we can adapt a policy to choose the best arm (test case) t_c from T_t to compose T'_t , then remove t_c from T_t , and continue this process until no more test cases are available in T_t . An arm t_c is chosen only once and the order of choice defines the

¹In this work we use *NAPFD* (see Section 4.5.1).

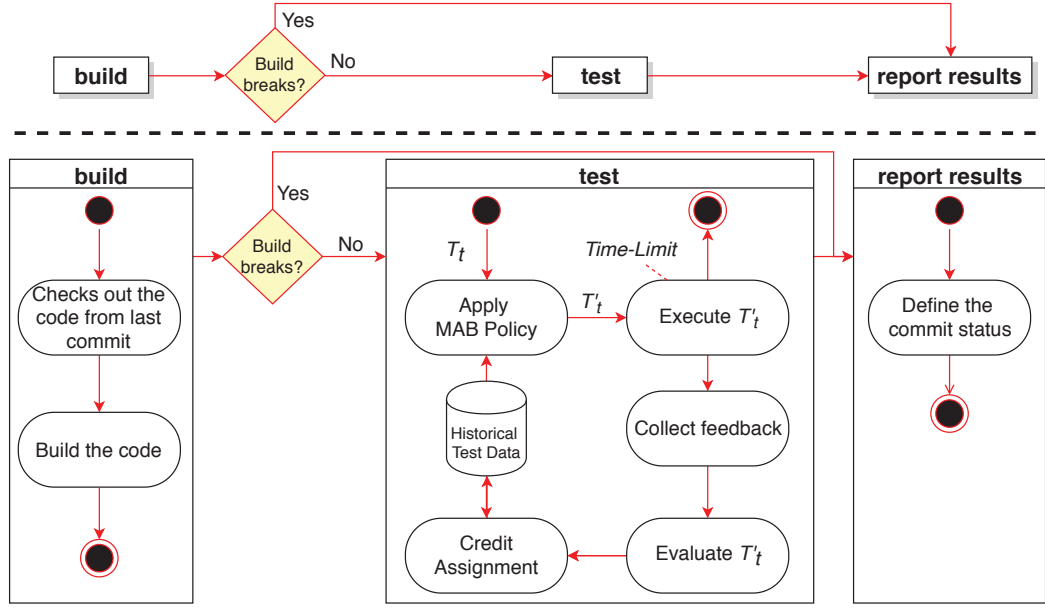


Figure 4.1: Overview of the proposed approach and how it is integrated in the test phase of a *CI* environment.

This figure presents in the top the main phases (activities) performed in a *CI* environment, and above these activities are detailed. In the test phase, the proposed approach is applied using the test case set T_t from the current commit and produces a prioritized test case set T'_t .

prioritization execution. But this process of choice is costly, once that a new evaluation for each test case in T_t is necessary to choose the next best test case to compose T'_t .

To reduce the selection cost, we adapted the policies to evaluate all the test cases (arms) at each time t . In this way, ordering the test cases, putting the best test case in the top, followed by the second best one, and so on. When more than one test case has the same *performance*, the order among them is defined randomly. This simple adaption allows a fast prioritization whilst considers the characteristics of the policy chosen. We also adapt the chosen policy to consider only the test cases available in time t and ignore the other ones from the previous time. This modification allows us to consider the dynamic environment (volatility) of the test cases.

As mentioned in Section 2.1, there are many *MAB* policies. We chose the policies that better work with the *EvE* dilemma: ϵ -greedy, *UCB*, and *FRRMAB*. Besides these policies, we also assessed the following policies as baselines: i) Random: is a strategy (not a real strategy per se) where the player will only do exploration; and ii) Greedy: only takes the best apparent arm, and it is a special case of ϵ -greedy where $\epsilon = 0$, i.e., it always does exploitation.

It is important to highlight that *FRRMAB* policy works with a sliding window. The reward value (*FIR* for *FRRMAB*) is obtained through a reward function (Section 4.3.1), then the last W rewards are used by the policy. In this way, for each test case, *FRRMAB* policy considers the history of rewards whilst the other ones use cumulative rewards.

4.3 CREDIT ASSIGNMENT

This procedure reflects the goal of the prioritization and teaches the *MAB*-based policy about the test cases considering historical test data. In this procedure, for $t'_c \forall t'_c \in T'_t$, two values are assigned: the number of times $n_{t'_c}$ has been applied before and the reward value ($\hat{q}_{t'_c,t}$). These values are used by the policies to generate the prioritized test set. At the beginning (where $t = 1$)

and for each new test case, the values for $n_{t'_c}$ and $\hat{q}_{t'_c,t}$ are assigned with zero. After that, the values are assigned as follows.

The number of times $n_{t'_c}$ that t'_c has been applied before is considered to explore new test cases (few used). Traditionally, a *MAB* policy selects an arm and increments the number of times that this arm was chosen. In our case, we use a combinatorial *MAB*, and this kind of *MAB* selects a set of arms (test cases). To counterbalance the order of choice, a weight is given for each $t'_c \in T'_t$ according to its order in T'_t . The weights are evenly spaced values within an interval (0.0, 1.0) with a step size of $\frac{1}{|T'_t|}$ in descending order. In this way, the highest weight is given to the first test case in T'_t and the lowest to the last one. Then, $n_{t'_c}$ is incremented with the weight defined to t'_c .

The reward value is obtained by a reward function (Section 4.3.1). This value is used to exploit the best test cases. As described in Section 4.2, the reward value is stored in a sliding window when *FRRMAB* policy is used whilst ϵ -greedy, greedy, and *UCB* policies use a cumulative reward strategy. If a new test case appears, a zero is set for the reward value, once that we do not have a *test case history*. On the other hand, if a test case is removed in the current cycle (commit), we remove its history.

4.3.1 Reward Functions

In this work, we adopt and adapted two reward functions from related work [90]. The first reward function *RNFail* (*Reward Based on Failures*) is based on the number of failures associated with a test case $t'_c \in T'_t$ and uses the function *fails* defined in Table 4.1.

Table 4.1: Functions used by the Reward Functions.

Definition	Description
$failurest'_c$	In our context, a test case t'_c can be composed by many parts (or test methods), each one of this part can be associated with a failure. In this way, a failing test case t'_c can be associated with one or more failures. Function $failurest'_c$ returns the number n_f of failures associated with t'_c .
$failst'_c$	The function $failst'_c$ returns 1 if $failurest'_c \geq 1$, and 0 otherwise.
$rankt'_c$	The function $rankt'_c$ returns the position of t'_c in a prioritized set T'_t .
$prect'_{c_1}, t'_{c_2}$	The function $prect'_{c_1}, t'_{c_2}$ returns 1 if $rankt'_{c_1} < rankt'_{c_2}$.

$$RNFailt'_c = \begin{cases} 1 & \text{if } failst'_c \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

The second reward function *TimeRank* (*Time-Ranked Reward*) is based on the rank of $t'_c \in T'_t$ (Equation 4.2). The idea is to evaluate whether failing test cases, with a greater number of failures, are ranked in the first positions in T'_t . To this end, a test case t'_c that does not fail and precedes failing test cases is penalized by their early scheduling.

$$TimeRankt'_c = \frac{|T'^{fail}| - \neg failst'_c \times \prod_{i=1}^{|T'^{fail}|} prect'_c, t'_{c_i}}{|T'^{fail}|} \quad (4.2)$$

where T'^{fail} is composed by the test cases of T'_t that failed. A non failed test case receives a reward given by the accumulated number of test cases which failed until its position in the

prioritization rank, that is, it receives a reward decreased by the number of failing test cases ranked after it in the rank.

4.4 COLEMAN'S ILLUSTRATIVE EXAMPLE

In order to provide a *COLEMAN*'s illustrative example, we created a small system that contains 5 commits (*CI* Cycles/Builds), 14 failures (all commits have at least one failing test case), and 8 tests that range between 4 and 7 tests across the commits. Figure 4.2 presents a heatmap concerning the test cases vs. failures along with *CI* cycles, where the axis X represents the *CI* Cycles and Y the all tests identified. In the matrix, the gray color in the intersection means the presence of a test in a *CI* cycle, and the transition from gray to black color represents the number of tests that failed in a *CI* cycle. Here, each test case reveals only one failure in each *CI* Cycle.

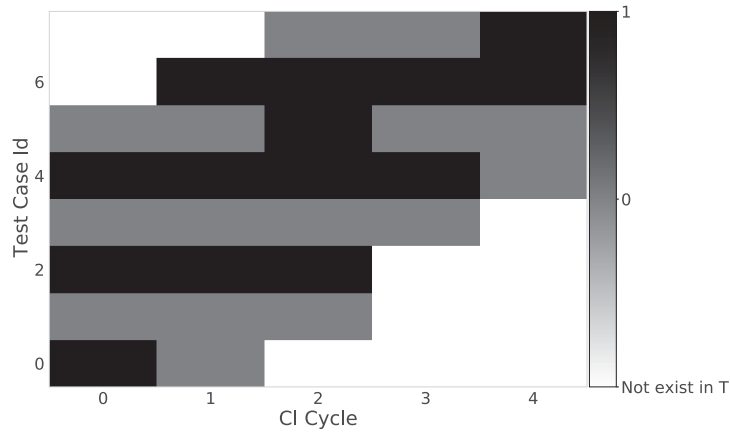


Figure 4.2: Heatmap concerning the test cases vs. faults along *CI* cycles from the System Example.

In our example, we use: *UCB* policy with $C = 0.3$, time budget of 50%, Time-ranked reward function, and the quality indicators *NAPFD* and *APFDc*. Figure 4.3 presents the behavior of the first commit ($t = 1$) inside *COLEMAN*. In this figure, the test case set available is composed by $T_1 = \{1, 2, 3, 4, 5, 6\}$, and in our example the failing test case set is $T_1^{fail} = \{1, 3, 5\}$. It is important to highlight that T_t^{fail} is unknown before the test execution.

After *COLEMAN* receives T_1 , it updates the arms to represent the available test cases, and then the prioritization is performed based on the information from previous prioritizations. For this, we use a *Q-table*, a lookup table used to calculate the maximum expected future rewards for action at each state, that guides us to the best action at each state.

In our example, each row in the *Q-table* represents a test case, and each column contains: the number of times, $n_{t'_c}$, a test case has been applied before (action attempts); the reward value $\hat{q}_{t'_c, t}$ (value estimates); and the quality estimate Q . The value estimate contains the accumulative reward according to the *MAB* policy used. For instance, *UCB* uses this value, while *FRRMAB* stores it in a sliding window. On the other hand, the quality estimate contains the evaluation made by a *MAB* policy. For instance, considering the *UCB* policy, the quality estimate is the result from Equation 2.2.

In the first commit, the prioritization is performed randomly because we do not have information to guide us during the prioritization process. For this reason, all values in the *Q-table* are filled with zeros, in which represents a new test cases. Then, the prioritized test set T'_1 is executed in the system (test execution) under a time constraint (test budget), the feedback is collected, and the prioritization quality is obtained. In the end, we apply the credit assignment to

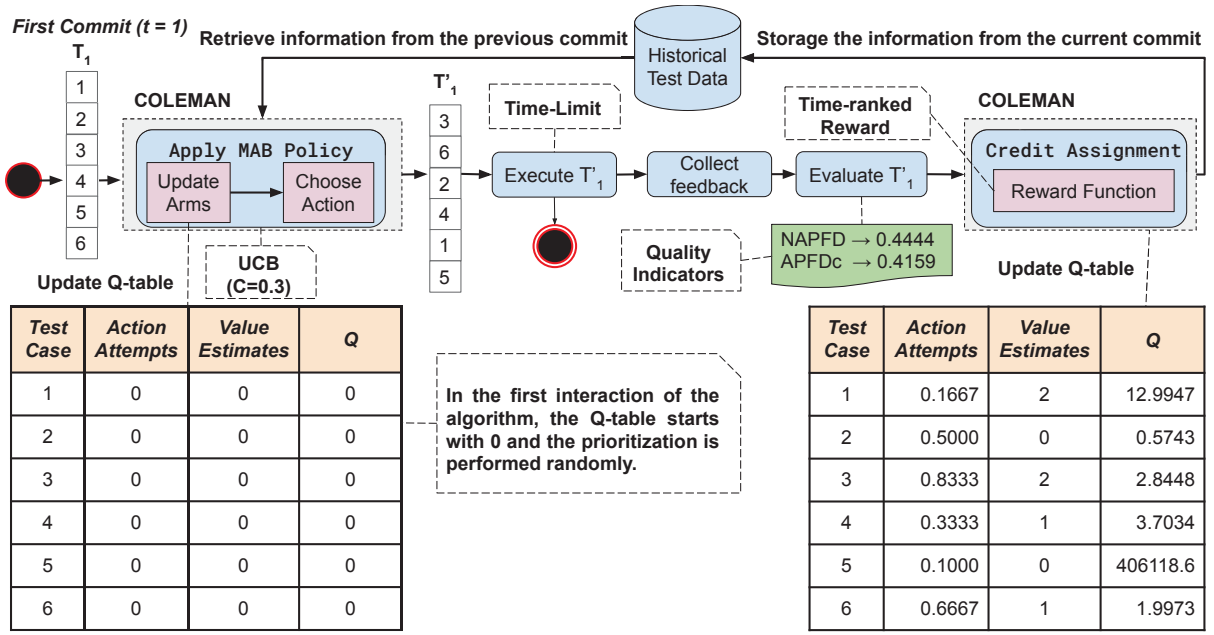


Figure 4.3: COLEMAN illustrative example - Commit 1.

calculate the rewards for each test. Then, Q -table is updated, and the historical test data is stored to be used by the policy in the subsequent commits.

In the second commit $t = 2$ (Figure 4.4), the test case available is composed by $T_2 = \{7, 6, 5, 2, 3, 1, 4\}$, and the failing test case set is $T_2^{fail} = \{7, 5, 3\}$. In this commit, the test case t_{c7} appeared, consequently, a new arm is added to the policy and a new row with zeros is added to Q -table. Such a behavior allow us to encompass the test case volatility.

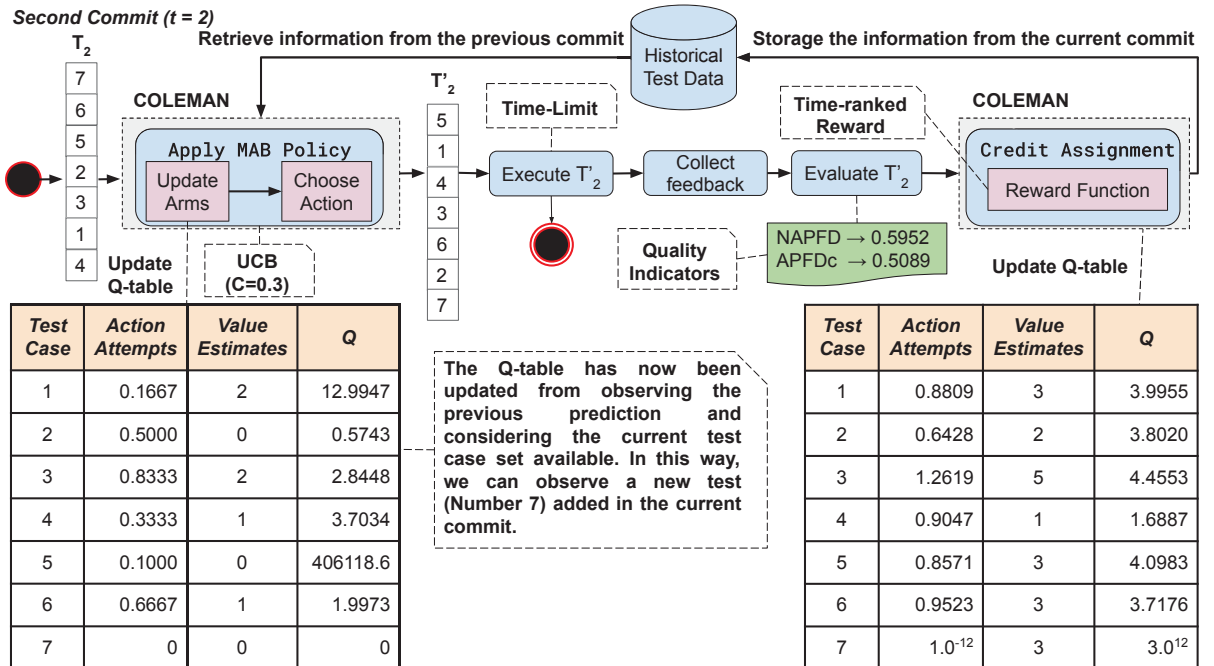


Figure 4.4: COLEMAN illustrative example - Commit 2.

The historical data from the previous prioritization is used to guide the current prioritization from the second commit. Then, T'_2 is provided by the MAB policy, in which is executed,

provide feedback, evaluated, and Q -table is updated at the end of the *COLEMAN*'s process. Such a behavior repeat for the subsequent commits.

In $t = 3$ (Figure 4.5), we have $T_3 = \{6, 8, 7, 5, 2, 3, 4\}$, and the failing test case set $T_3^{fail} = \{6, 7, 5, 3\}$. In this commit, the test t_{c8} was added and the test t_{c1} was removed.

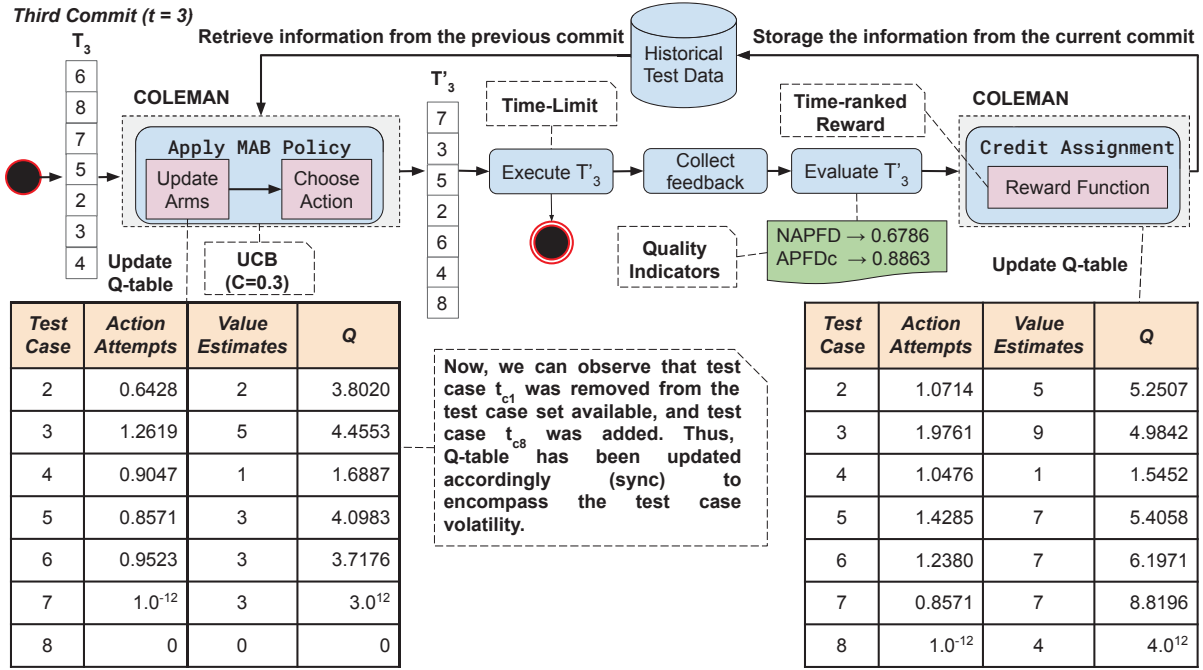


Figure 4.5: COLEMAN illustrative example - Commit 3.

In $t = 4$ (Figure 4.6), we have $T_4 = \{4, 5, 6, 7, 8\}$ and $T_4^{fail} = \{5, 7\}$. In this commit, the tests t_{c2} and t_{c3} were removed. In the last commit $t = 5$ (Figure 4.7), the test set available is composed by $T_5 = \{6, 8, 7, 5\}$ and $T_5^{fail} = \{8, 7\}$. In this commit, the test t_{c4} was removed.

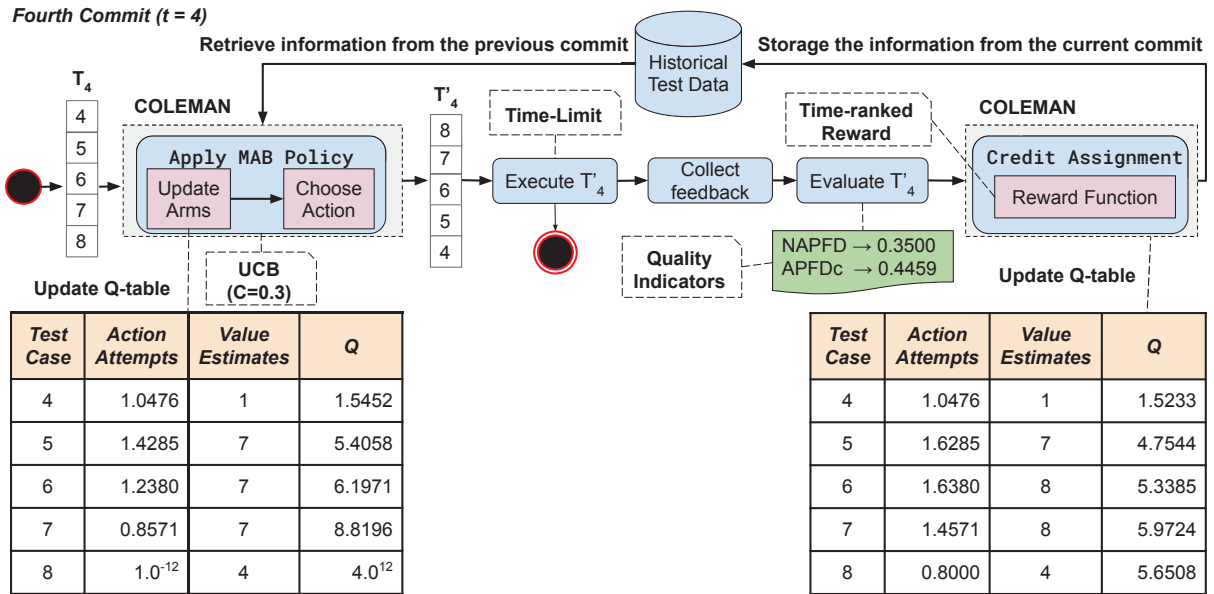


Figure 4.6: COLEMAN illustrative example - Commit 4.

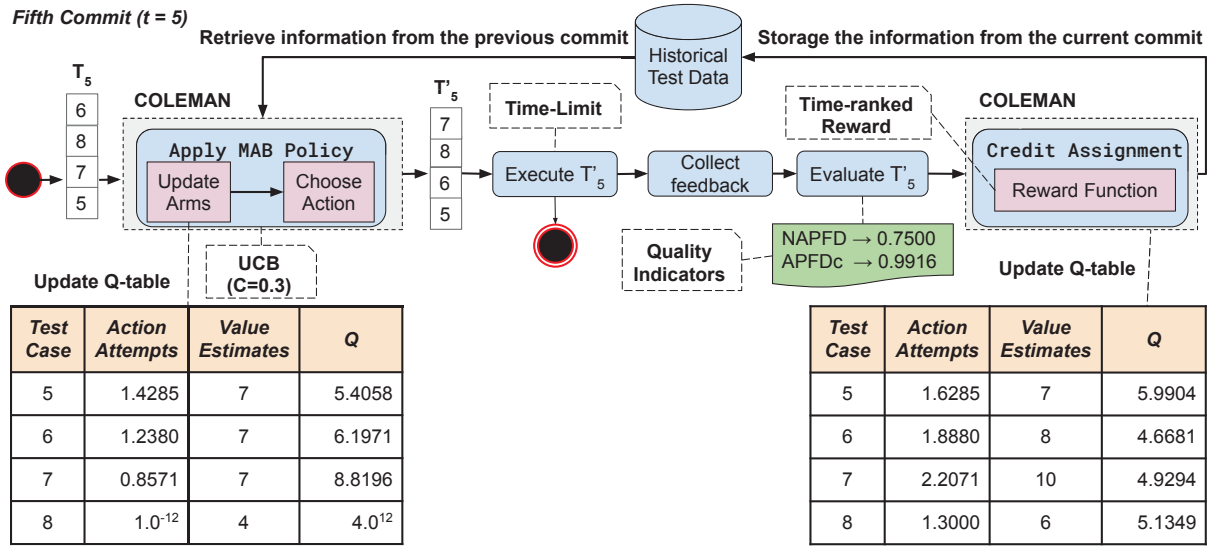


Figure 4.7: COLEMAN illustrative example - Commit 5.

This illustrative example shows how *COLEMAN* acts in a system during the CI environment. We can see how easy is to keep the historical information required by a *MAB* policy, and we can test manually a *MAB* policy to assess the quality if needed.

4.5 EVALUATION DESCRIPTION

The main hypothesis of this work is that *MAB* can be used to address the *TCPCI* problem in a very cost-effective way. Then, our experiment evaluates the *COLEMAN* applicability and performance in *CI* environments. We also perform a comparison with related work. The experiment is guided by the following research questions:

- RQ1:** *What is the best configuration for COLEMAN?* This question aims to identify the best *MAB* policy and reward function to be used with *COLEMAN*.
- RQ2:** *Is COLEMAN applicable in the CI development context?* This question is specially important for software testers that want to use *COLEMAN* in practice. It investigates whether the time spent in the prioritization is acceptable considering *CI* cycles (commits).
- RQ3:** *Can COLEMAN outperform RETECS?* This question compares our approach, with *RETECS*, the *RL* approach, which is the most similar to *COLEMAN*.

To answer RQ1, we compare five *MAB* policies: Random, ϵ -Greedy, Greedy, *UCB*, and *FRRMAB*, (Section 4.2) and both reward functions: *RNFail* and *TimeRank* (Section 4.3.1). We use indicators commonly applied in *TCP* (Section 4.5.1). We evaluate three time constraints (budgets) considering: 10%, 50%, and 80% of the execution time of the overall test set available in each commit. The best policy identified in RQ1 is used to answer the remaining questions.

To answer RQ2, we compare the prioritization time spent by *COLEMAN* and the time between commits, as well as the percentage of reduced time to test execution.

To answer RQ3, we execute the implementation of *RETECS* available in the literature² by using *ANN*, which obtained the best results in comparison with a Tableau representation [90]. Our approach considers only a minimal information to prioritize the test cases, historical failure

²<https://bitbucket.org/helges/RETECS>

data, whilst *RETECS* needs additional information concerning each test case: duration, the time it was last executed, and results from its previous execution (passed or failed).

In this experiment, if a test case is removed in a commit, it is then removed along with its history. The results are obtained from 30 independent executions for each system, reward function, and time budget. All the experiments are performed on an Intel® Xeon® E5-2640 v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS.

4.5.1 Quality Indicators

To evaluate the performance of the approaches concerning failure detection effectiveness of a test suite T , we use *NAPFD* [81] and *APFDc* [21]. See Chapter 2.3 for further details.

To evaluate the test suite efficiency concerning how fast it is to detect a fault, we use the *RFTC*. In this rank, lower values represent a faster failure detection. In this sense, we extract the order of the first test case that fails from the prioritized test suite. This metric is useful when we need fast feedback from test cases and there is only one fault that the test cases are seeking.

Furthermore, we define a metric (Equation 4.3) named *NTR*, in order to observe the difference between time spent until the first test case fails r_t and the total time spent to execute all tests \hat{r}_t . In this metric, only the commits that failed, CI^{fail} , are considered. In this way, we can evaluate the capability of an algorithm to reduce the time spent in a *CI* cycle.

$$NTRA = \frac{\sum_{t=1}^{CI^{fail}} \hat{r}_t - r_t}{\sum_{t=1}^{CI^{fail}} \hat{r}_t} \quad (4.3)$$

The last indicator used is Prioritization Time. Such measure computes the time spent (in seconds) by an algorithm to perform the prioritization. This value helps to observe whether an approach spends much time, what can make it impracticable for real scenarios.

We apply Kruskal-Wallis [46], Mann-Whitney [54], and Friedman [29] statistical tests with a confidence level of 95%. We use Kruskal-Wallis to evaluate the performance of the approaches in each system over 30 independent runs. We use Mann-Whitney to evaluate a pair of performances in the same system or for post-hoc analysis. We use Friedman to evaluate the approach behavior across different systems. To this end, each system becomes a dependent variable, in which we apply multiple approaches.

Additionally, to calculate the effect size magnitude of the difference between two groups, we use the Vargha and Delaney's \hat{A}_{12} [94] metric. This measure ranges from 0 to 1 and defines the probability of a value, taken randomly from the first sample, is higher than a value taken randomly from the second sample. A *Negligible* magnitude $\hat{A}_{12} < 0.56$ represents a very small difference among the values and usually does not yield statistical difference. The *Small* $0.56 \leq \hat{A}_{12} < 0.64$ and *Medium* $0.64 \leq \hat{A}_{12} < 0.71$ magnitudes represent small and medium differences among the values, and may or not yield statistical differences. Finally, a *Large* magnitude $0.71 \leq \hat{A}_{12}$ represents a significantly large difference that usually can be seen in the numbers without much effort.

4.5.2 Systems Under Test

We select non-toy, non-fork, and active GitHub (GH) projects considering *watchers* and *stars* on GH, as well as some systems already used in the literature [37, 90]. Most of them using Travis CI [93] and Maven.³ To collect the Travis *CI* build history, we adapt and use TravisTorrent⁴ [4].

Each *System Under Test* (*SUT*) is detailed in Table 4.2. The second column shows the period of build logs analyzed. The third column presents the total of builds identified, and in parenthesis the number of builds included in the analysis. We discard build logs with some problem, identified by Travis CI, such as the ones related to non-valid build logs and test cases that did not execute. The fourth column shows the total of failures found, and in parenthesis the number of builds in which at least one test failed. The fifth column shows the number of different (unique) test cases identified from build logs, and in parenthesis the range of test cases executed in the builds. The last columns present, for each system, the mean (\pm standard deviation) duration in minutes of the *CI* cycles and the interval between them.

Table 4.2: Test Case Set Information.

Name	Period	Builds	Failures	Test Cases	Duration (min)	Interval (min)
Druid	2016/04/24-2016/11/08	286 (168)	270 (71)	2391 (1778-1910)	4.27 \pm 10.66	384.76 \pm 468.86
Fastjson	2016/04/15-2018/12/04	2710 (2371)	940 (323)	2416 (900-2102)	1.97 \pm 0.89	233.22 \pm 401.26
Deeplearning4j	2014/02/22-2016/01/01	3410 (483)	777 (323)	117 (1-52)	12.33 \pm 14.91	306.05 \pm 442.55
DSpace	2013/10/16-2019/01/08	6309 (5673)	13413 (387)	211 (16-136)	11.78 \pm 7.03	291.29 \pm 411.19
Guava	2014/11/06-2018/12/02	2011 (1689)	7659 (112)	568 (308-512)	62.53 \pm 80.31	435.55 \pm 464.52
OkHttp	2013/03/26-2018/05/30	9919 (6215)	9586 (1408)	289 (2-75)	7.64 \pm 5.64	220.17 \pm 405.93
Retrofit	2013/02/17-2018/11/26	3719 (2711)	611 (125)	206 (5-75)	2.40 \pm 1.60	270.86 \pm 449.41
ZXing	2014/01/17-2017/04/16	961 (605)	68 (11)	124 (81-123)	13.14 \pm 12.37	411.10 \pm 465.53
IOF/ROL	2015/02/13-2016/10/25	2392 (2392)	9289 (1627)	1941 (1-707)	1537.27 \pm 2018.73	1324.26 \pm 291.75
Paint Control	2016/01/12-2016/12/20	20711 (20711)	4956 (1980)	1980 (1-74)	424.46 \pm 275.90	1417.86 \pm 144.97
GSDTSR	2016/01/02-2016/02/01	259388 (259388)	3208 (2924)	5555 (1-390)	974.25 \pm 4850.66	1439.91 \pm 2.58

Druid is a database connection pool written in Java used by Alibaba. Fastjson, created by Alibaba, is a Java library that can be used to a fast JSON parser/generator for Java. Deeplearning4j is a deep learning library for Java Virtual Machine. DSpace is an open source software that provides facilities for the management of digital collections, used for the implementation of institutional repositories. Guava is a set of core libraries for Java, developed by Google, which includes new collection types, APIs/utilities for concurrency, I/O, and others. OkHttp, developed by Square, is an HTTP and HTTP/2 client for Android and Java applications. Retrofit, also developed by Square, is a type-safe HTTP client for Android and Java. ZXing (*Zebra Crossing*) is a barcode scanning library for Java and Android. The systems IOF/ROL and Paint Control are industrial datasets for testing complex industrial robots from ABB Robotic⁵. GSDTSR is *The Google Dataset of Testing Results* [22] with a sample of 3.5 million test suite execution results from Google products.

The systems IOF/ROL, Paint Control, and GSDTSR are selected for comparison because they are the same systems used in related work to evaluate *RETECS* [90]. However, in the related work the datasets are analyzed considering that a *CI* cycle includes all the test cases executed per day. In our study, we consider the *CI* cycle as a commit. In this way, we change the datasets representations to consider each date in the last run information is a commit. These systems have different characteristics (number of faults, test cases, and commits), and they can ensure the evaluation of the approaches in relation to the generalization capacity.

³Maven is a build automation tool used primarily for Java projects. We choose projects which use Maven as a testing framework because it provides detailed output traces (more verbose).

⁴<https://github.com/jacksonpradolima/travistorrent-tools>

⁵<https://new.abb.com/products/robotics>

4.5.3 Data Collection

In this work, we followed some steps (Figure 4.8) to identify relevant systems, extract information from them, and model datasets with the data collected⁶. In the first phase we identify relevant projects (systems). For this, we made use of the GHTorrent [32], which provides a GitHub REST API, to identify relevant Java projects. We identified 16852 projects. After, we filtered en-mass whether a project has a Travis CI [93] build history. For this, we used TRAVIS POKER tool from Travis Torrent, and we found 4128 out of 16852 projects. Due to a large number of projects, we filtered the projects that use Maven and selected a subset of 8 projects, which contain different characteristics concerning the number of build logs and test cases, as well as the systems already used in the literature.

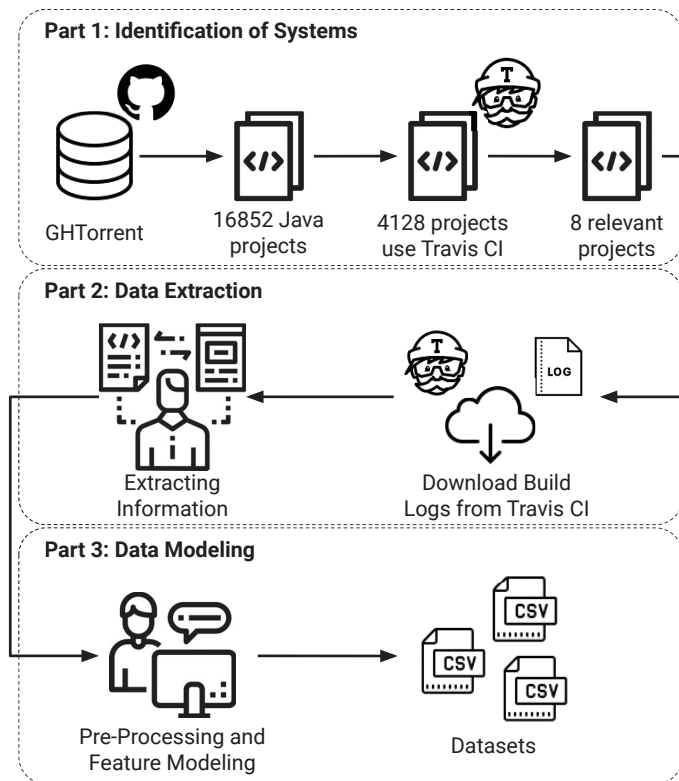


Figure 4.8: Steps conducted for extracting and modeling build history from Travis CI.

We chose Java systems that use Maven because they provide in the log execution (build log⁷) adequate information about the list of executed test cases and their status (executed and skipped test cases, errors, and time spent). This can help to minimize the effort in data extraction phase, once that we analyzed the build logs, and their structures and formats vary among software projects, depending on the build and testing framework used in the development.

The second phase of our experimental framework aims at finding and extracting relevant information from Travis CI build history. So, for each system chosen, we downloaded detailed information from Travis CI build history using TRAVIS HARVESTER from Travis Torrent. Then, we applied the BUILDLOG ANALYZER from Travis Torrent that parses build logs and searched for output traces. A testing framework used in a system can produce different outputs. Due to this, we chose projects which use Maven as a testing framework which provides detailed output traces (more verbose).

⁶The mining of the software repositories was performed on December, 2018.

⁷A build log contains the log of a *CI* cycle.

On the third phase, we employed feature modeling to organize the data collected. In this way, we ran a pre-processing to identify and remove any noise in the build logs. During the collection, we identified some noises in the build logs concerning duplicated information about test cases. For these cases, we preserved the greatest values. We also extracted information about the failures detected along the *CI* cycles and test case volatility. Figure 4.9(a) shows an example of this information for the Deeplearning4j system, in which can be used to investigate any data correlation with the results found by the experiment. On the other hand, Figure 4.9(b) presents heatmap concerning the test cases vs. failures along *CI* cycles for Deeplearning4j system, and this figure can be used to identify the test case volatility, as well as to identify if a test case which appears reveal a failure. In this figure, the gray color represents the presence of a test case in a *CI* cycle. The transition from gray to black color represents the number of tests that failed in a *CI* cycle.

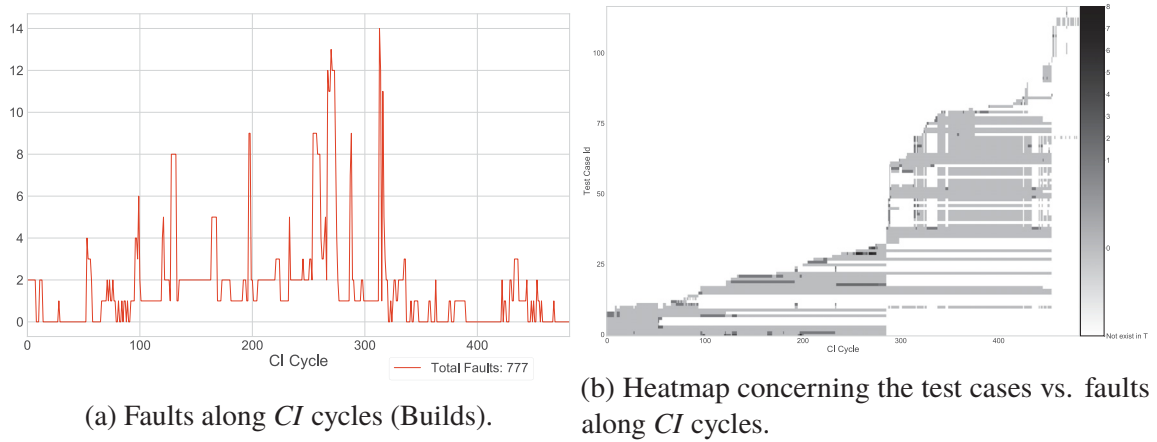


Figure 4.9: Information extracted from Deeplearning4j system behavior.

In the end, the information extracted and selected were converted to a individual CSV file for each system, including: Id: unique numeric identifier of the test execution; Name: unique numeric identifier of the test case; BuildId: a value uniquely identifying the build; Duration: approximated runtime of the test case; LastRun: previous last execution of the test case as *DateTime*; NumRan: number of executed test cases; Errors: number of errors revealed; Verdict: test verdict of this test execution (Failed: 1, Passed: 0).

4.5.4 Parameters Setting

RETECS is available online and it is evaluated using *ANN* with the default values defined in related work [90], for Hidden Nodes, Replay Memory, and Replay Batch Size, with, respectively, 12, 10000, and 1000. In this way, a tuning phase was necessary only to choose the parameters values for the *MAB* policies *UCB*, *FRRMAB*, and ϵ -Greedy. They are the scaling factor C to control the trade-off among *EvE* for *UCB* and *FRRMAB* policies, the sliding window size W and decayed factor DF for *FRRMAB*, and the probability ϵ for ϵ -Greedy policy. The possible values used in this phase for C are 0.3, 0.5, 1.0, and 2.0, for W were 10, 50, 100, 150, 200, 250, and 300, and for ϵ 0.1, 0.2, 0.5, 0.7, and 0.9. After some empirical studies, the decayed factor is no longer used and a default value equals to 1 is defined.

We conduct an empirical evaluation with 10 independent runs for each *MAB* policy with different parameters. For this evaluation, we consider the systems Deeplearning4j and Fastjson. These systems are chosen because they have the best trade-off between the mean number of failures by *CI* cycles and the number of *CI* cycles. Additionally, we define a test

budget for a *CI* cycle with a fixed percentage of 50% of the required time as adopted in [90]. The best parameters found for the *MAB* policies are: ϵ -Greedy and *UCB* with 0.5 (in the *TimeRank* function) and 0.3 (*RNFail*) for parameter C ; and *FRRMAB* with 0.3 for C and 100 for W .

4.5.5 Threats to Validity

We identify the following points that can be threats to the validity of the results. The first threat is the parameter configuration of the algorithms. Other parameters can lead to different results. To mitigate this threat, we empirically evaluate different ranges of parameters for the *MAB* policies and for *RETECS* we adopt the configuration of related work, since we are using the same systems.

The dataset representation used is a threat. We change information about the *CI* cycles for systems IOF/ROL, Paint Control, and GSDTSR. This change can impact the results, mainly because we do not know with precision what information is from each commit and whether *RETECS* is deeply dependent on this kind of representation. For this reason, we group the data by the last run date.

4.6 RESULTS AND ANALYSES

In this section, the experimental results are presented and analyzed aiming to answer the posed questions. Supplementary material with datasets, results, and additional analysis can be found in our public repository [70].

4.6.1 RQ1: *COLEMAN* Configuration

As mentioned before, to answer RQ1, we compare the five *MAB* policies with both reward functions taking into account three budgets. We use the indicators: *NAPFD*, *APFD_c*, and *RFTC*. The results and a complete analysis are available in supplementary material [70]. Next, we detail only *NAPFD* and *APFD_c* results regarding the budget of 50% (Table 4.3). According to Spieker et al. [90], a time budget of 50% presents a constraint that allows better comparison whilst keeps the difficulty inherent from the problem.

The average is computed using results from 30 independent executions found by each policy in each *SUT*. Values highlighted in bold are the best, and values that are statistically equivalent to the best ones have their corresponding cells painted in light gray. Furthermore, we use different symbols to indicate the effect size magnitude concerning the best values. For each comparison in each *SUT*, we use the Kruskal-Wallis test. When detected statistical difference, we apply a post-hoc analysis using the Mann-Whitney test with Bonferroni *p-value* adjustment method to find the statistical difference among the policies. In order to evaluate the performance of the policy concerning all systems, we compute the average across the systems. The average values obtained are shown in the last line in each table. We compare these values using Friedman.

Regarding *NAPFD*, Table 4.3 shows *FRRMAB* stands out in 68% of the cases (15 out of 22 cases, considering 11 systems and 2 functions). Such cases are represented by grey cells in the table. *FRRMAB* is followed by *UCB* with 50% of the cases (11 out of 22). *FRRMAB* is the best with statistical difference over the others in 8 cases. *UCB* is the best only in one case for *RNFail*. The effect size results endorse what we observed in the numbers. Besides that, when there is a statistical difference among policies, the effect size tends to be *large*. The performance of these policies is similar for both functions. For *RNFail*, *FRRMAB* stands out in 7 cases (out of 11), and *UCB* stands out in 5 (out of 11). For *TimeRank*, *FRRMAB* stands out in 8, and *UCB* stands out in 6. Overall, we observe that for *RNFail* the other *MAB* policies perform better. For this reward function, more policies have values that are close to the best one. Among the *MAB*

Table 4.3: NAPFD and APFDc values (mean and standard deviation) for *MAB* policies using time budget of 50%.

This table reports the *NAPFD* and *APFDc* results (averages \pm standard deviation) obtained from 30 independent runs with time budget of 50% and organized by each Reward Function under study (see Section 4.3.1). Values highlighted in bold with a “★” symbol denotes the best algorithm for a Reward Function in a *SUT* and, in gray, results that are statistically equal to the best one. A “▼” indicates that the effect size was negligible in relation to the best, while “▽” denotes a small magnitude, “△” a medium magnitude, and “▲” a large magnitude. The effect size was performed during the post-hoc tests, that is, when there is a statistical difference.

SUT	NAPFD					APFDc				
	FRRMAB	UCB	c-Greedy	Greedy	Random	FRRMAB	UCB	c-Greedy	Greedy	Random
RNFail - REWARD BASED ON FAILURES										
Druid	0.9333 \pm 0.013 △	0.9422 \pm 0.007 ★	0.8472 \pm 0.110 ▲	0.8965 \pm 0.072 ▽	0.7464 \pm 0.014 ▲	0.9486 \pm 0.016 ★	0.9486 \pm 0.007 ▼	0.8477 \pm 0.110 ▲	0.8979 \pm 0.073 △	0.7506 \pm 0.014 ▲
Fastjson	0.9174 \pm 0.021 ▲	0.9597 \pm 0.001 ★	0.9501 \pm 0.005 ▲	0.9507 \pm 0.005 ▲	0.9176 \pm 0.003 ▲	0.9186 \pm 0.021 ▲	0.9595 \pm 0.001 ★	0.9491 \pm 0.006 ▲	0.9498 \pm 0.005 ▲	0.9193 \pm 0.002 ▲
DeepLearning4j	0.7890 \pm 0.001 ▲	0.7911 \pm 0.002 ▲	0.8066 \pm 0.003 △	0.8084 \pm 0.002 ★	0.6381 \pm 0.011 ▲	0.8106 \pm 0.001 ★	0.7971 \pm 0.002 ▲	0.7974 \pm 0.004 ▲	0.7961 \pm 0.003 ▲	0.6404 \pm 0.016 ▲
DSpace	0.9724 \pm 0.009 ★	0.9720 \pm 0.001 ▼	0.9692 \pm 0.002 ▼	0.9685 \pm 0.002 ▼	0.9581 \pm 0.001 ▲	0.9737 \pm 0.009 ★	0.9730 \pm 0.001 ▼	0.9713 \pm 0.002 ▼	0.9704 \pm 0.002 ▼	0.9587 \pm 0.001 ▲
Guava	0.9653 \pm 0.004 ▲	0.9750 \pm 0.002 △	0.9768 \pm 0.006 ★	0.9761 \pm 0.010 ▼	0.9603 \pm 0.002 ▲	0.9756 \pm 0.003 ▲	0.9756 \pm 0.002 ▼	0.9770 \pm 0.006 ★	0.9758 \pm 0.010 ▼	0.9611 \pm 0.002 ▲
OkHttp	0.9192 \pm 0.000 ★	0.9023 \pm 0.001 ▲	0.9041 \pm 0.005 ▲	0.8993 \pm 0.006 ▲	0.8513 \pm 0.002 ▲	0.9177 \pm 0.000 ★	0.9039 \pm 0.002 ▲	0.8994 \pm 0.005 ▲	0.8950 \pm 0.006 ▲	0.8549 \pm 0.002 ▲
Retrofit	0.9853 \pm 0.000 ★	0.9799 \pm 0.001 ▲	0.9717 \pm 0.001 ▲	0.9717 \pm 0.001 ▲	0.9710 \pm 0.002 ▲	0.9850 \pm 0.000 ★	0.9794 \pm 0.001 ▲	0.9722 \pm 0.001 ▲	0.9722 \pm 0.001 ▲	0.9712 \pm 0.001 ▲
ZXing	0.9846 \pm 0.000 ▲	0.9876 \pm 0.000 ▲	0.9873 \pm 0.002 ▲	0.9869 \pm 0.002 ▲	0.9892 \pm 0.002 ★	0.9862 \pm 0.000 ▲	0.9877 \pm 0.000 ▲	0.9876 \pm 0.001 ▲	0.9873 \pm 0.001 ▲	0.9893 \pm 0.001 ★
IOF/ROL	0.5046 \pm 0.002 ★	0.4791 \pm 0.003 ▲	0.4792 \pm 0.002 ▲	0.4790 \pm 0.002 ▲	0.4786 \pm 0.002 ▲	0.5081 \pm 0.002 ★	0.4794 \pm 0.003 ▲	0.4796 \pm 0.002 ▲	0.4793 \pm 0.002 ▲	0.4788 \pm 0.002 ▲
Paint Control	0.9150 \pm 0.000 ★	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9162 \pm 0.000 ★	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲
GSDTSR	0.9893 \pm 0.000 ★	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9894 \pm 0.000 ★	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9892 \pm 0.000 ▲
Average	0.8978	0.8993	0.8905	0.8946	0.8558	0.9021	0.9007	0.8895	0.8934	0.8571
TimeRank - TIME-RANKED REWARD										
Druid	0.9710 \pm 0.008 ★	0.8422 \pm 0.066 ▲	0.8767 \pm 0.040 ▲	0.8503 \pm 0.048 ▲	0.7489 \pm 0.014 ▲	0.9787 \pm 0.009 ★	0.8768 \pm 0.078 ▲	0.8988 \pm 0.042 ▲	0.8871 \pm 0.056 ▲	0.7534 \pm 0.014 ▲
Fastjson	0.9118 \pm 0.028 ▲	0.9544 \pm 0.002 ★	0.9455 \pm 0.004 ▲	0.9240 \pm 0.009 ▲	0.9181 \pm 0.003 ▲	0.9140 \pm 0.027 ▲	0.9565 \pm 0.002 ★	0.9516 \pm 0.004 ▲	0.9303 \pm 0.009 ▲	0.9199 \pm 0.003 ▲
DeepLearning4j	0.8200 \pm 0.000 ★	0.7193 \pm 0.002 ▲	0.7028 \pm 0.004 ▲	0.7047 \pm 0.002 ▲	0.6366 \pm 0.007 ▲	0.8134 \pm 0.001 ★	0.7879 \pm 0.004 ▲	0.7774 \pm 0.007 ▲	0.7805 \pm 0.004 ▲	0.6405 \pm 0.012 ▲
DSpace	0.9766 \pm 0.008 ★	0.9659 \pm 0.001 ▲	0.9606 \pm 0.002 ▲	0.9602 \pm 0.002 ▲	0.9582 \pm 0.001 ▲	0.9767 \pm 0.009 ★	0.9683 \pm 0.001 ▲	0.9646 \pm 0.002 ▲	0.9649 \pm 0.002 ▲	0.9588 \pm 0.001 ▲
Guava	0.9675 \pm 0.007 ▲	0.9698 \pm 0.002 ▲	0.9738 \pm 0.002 ★	0.9734 \pm 0.004 ▼	0.9608 \pm 0.002 ▲	0.9672 \pm 0.007 ▲	0.9740 \pm 0.003 ▲	0.9786 \pm 0.003 ▲	0.9824 \pm 0.004 ★	0.9614 \pm 0.002 ▲
OkHttp	0.9317 \pm 0.000 ★	0.8753 \pm 0.001 ▲	0.8725 \pm 0.002 ▲	0.8677 \pm 0.004 ▲	0.8514 \pm 0.002 ▲	0.9246 \pm 0.000 ★	0.8930 \pm 0.001 ▲	0.8881 \pm 0.002 ▲	0.8866 \pm 0.004 ▲	0.8550 \pm 0.002 ▲
Retrofit	0.9893 \pm 0.000 ★	0.9789 \pm 0.001 ▲	0.9715 \pm 0.002 ▲	0.9689 \pm 0.001 ▲	0.9707 \pm 0.001 ▲	0.9885 \pm 0.000 ★	0.9798 \pm 0.001 ▲	0.9733 \pm 0.002 ▲	0.9712 \pm 0.001 ▲	0.9706 \pm 0.001 ▲
ZXing	0.9857 \pm 0.000 ▲	0.9879 \pm 0.001 ▲	0.9882 \pm 0.002 ▼	0.9861 \pm 0.001 ▲	0.9891 \pm 0.001 ★	0.9869 \pm 0.000 ▲	0.9880 \pm 0.001 ▲	0.9882 \pm 0.002 ▲	0.9861 \pm 0.001 ▲	0.9894 \pm 0.001 ★
IOF/ROL	0.5189 \pm 0.002 ★	0.4787 \pm 0.002 ▲	0.4789 \pm 0.002 ▲	0.4786 \pm 0.002 ▲	0.4785 \pm 0.002 ▲	0.5223 \pm 0.002 ★	0.4789 \pm 0.002 ▲	0.4792 \pm 0.002 ▲	0.4789 \pm 0.002 ▲	0.4786 \pm 0.002 ▲
Paint Control	0.9150 \pm 0.000 ★	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9162 \pm 0.000 ★	0.9146 \pm 0.000 ▲	0.9146 \pm 0.000 ▲	0.9145 \pm 0.000 ▲	0.9145 \pm 0.000 ▲
GSDTSR	0.9894 \pm 0.000 ★	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9894 \pm 0.000 ★	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲	0.9891 \pm 0.000 ▲
Average	0.9070	0.8796	0.8795	0.8743	0.8560	0.9071	0.8915	0.8912	0.8883	0.8574

policies, the Random policy has the worst performance. Regarding *APFDc* we have similar findings. *FRRMAB* stands out in 72% of the cases (16 out of 22), and *UCB* in 45% of the cases (10 out of 22).

For the other budgets of 10% and 80%, we observed similar results regarding the performance of the policies and functions. As expected, the values for all policies are better for the budget of 80%, which is less restrictive. Considering *NAPFD* and all the 33 cases involving the three budgets (11 systems \times 3 time budgets), *FRRMAB* stands out the other policies in \approx 76% of the cases (25 out of 33) and 82% (27 cases), respectively, for the *RNFail* and *TimeRank* functions. *FRRMAB* is the best policy with statistical difference in \approx 40% of the cases (13 out of 33) for both reward functions, and *UCB* policy is the best in \approx 6% of the cases (2) and \approx 3% of the cases (1), respectively, for the *RNFail* and *TimeRank* functions. *FRRMAB* also obtains the best *APFDc* values and stands out in \approx 82% of the cases (27 out of 33) for the *RNFail* function and in \approx 76% of the cases (25) for the *TimeRank* function.

For *IOF/ROL*, the approach had the worst performance. This system has a high test case volatility and a high number of peaks of failure detection. In this way, it is an example of that to find a solution to the *TCPCI* problem is hard. We observe that the Random policy is defeated by *MAB* policies in almost all systems, except in *ZXing* in both reward functions. To understand this behavior, we analyze the failures detected over the cycles. In this system, we verify peaks in the failure detection in a few commits and long periods without failures. This scenario endorses our approach behavior once that it is based on historical test data. Furthermore, this system has a low test case volatility.

RFTC results are available in supplementary material [70]. We observe that the *NAPFD* average values found in each independent execution by reward functions and the early fault detection given by *RFTC* are, in most of the cases, correlated, that is, good *NAPFD* values provide good *RFTC* values. In most of the systems, the *MAB* policies are better than Random

policy, as well as they are more stable (low dispersion of values). Among the *MAB* policies, *FRRMAB* is more stable than the other ones. This shows that the use of a sliding window is interesting when a system contains peaks of detecting faults and periods of stability, as well as when there is a large number of commits.

RQ1: Results show *FRRMAB* is the best *MAB* policy, regarding *NAPFD*, *APFDc* values, and early fault detection given by the indicator *RFTC*. This happens for both reward functions, and the three budgets evaluated. As expected, better indicator values are obtained for the less restrictive budget of 80%. This happens for all policies. In particular, the combination of *FRRMAB* with *TimeRank* function provides better performance.

4.6.2 RQ2: COLEMAN Applicability

To answer RQ2, we use Table 4.4. The second and third columns presents, respectively, the mean prioritization time and standard deviation of *COLEMAN* with the *FRRMAB* policy (the best policy found in the analysis of RQ1), and time budget of 50%. In such a table, we can observe the time spent is negligible in most of the systems. A great time is spent in *Druid* and *Fastjson*, systems that also have a great number of test cases in a *CI* cycle. If we take the standard deviation and the worst case (*Fastjson*), *COLEMAN* takes less than one second to execute. The mean time spent with both reward functions is similar, as well as we do not observe any impact on the other budgets.

Table 4.4: Mean and standard deviation Prioritization Time (in seconds) with time budget of 50%.

SUT	PRIORITIZATION TIME (SEC.)			
	FRRMAB		ANN	
	RNFail	TimeRank	RNFail	TimeRank
Druid	0.2474 ± 0.040	0.2373 ± 0.041	0.3881 ± 0.268	0.3844 ± 0.279
Fastjson	0.3916 ± 0.191	0.3879 ± 0.191	0.2474 ± 1.382	0.2395 ± 1.288
Deeplearning4j	0.0271 ± 0.002	0.0271 ± 0.002	0.0038 ± 0.006	0.0187 ± 0.107
DSpace	0.0287 ± 0.002	0.0287 ± 0.002	0.0101 ± 0.042	0.0507 ± 0.186
Guava	0.0609 ± 0.016	0.0609 ± 0.016	0.0335 ± 0.039	0.0405 ± 0.066
OkHttp	0.0283 ± 0.002	0.0283 ± 0.002	0.0079 ± 0.022	0.0371 ± 0.125
Retrofit	0.0277 ± 0.002	0.0277 ± 0.002	0.0050 ± 0.010	0.0178 ± 0.079
ZXing	0.0343 ± 0.003	0.0343 ± 0.003	0.0156 ± 0.025	0.0229 ± 0.069
IOF/ROL	0.0278 ± 0.005	0.0278 ± 0.005	0.0034 ± 0.009	0.5364 ± 1.221
Paint Control	0.0256 ± 0.002	0.0256 ± 0.002	0.0020 ± 0.005	0.0028 ± 0.028
GSDTSR	0.0253 ± 0.002	0.0253 ± 0.002	0.0027 ± 0.014	0.0028 ± 0.013
Average	0.0841	0.0828	0.0654	0.1231

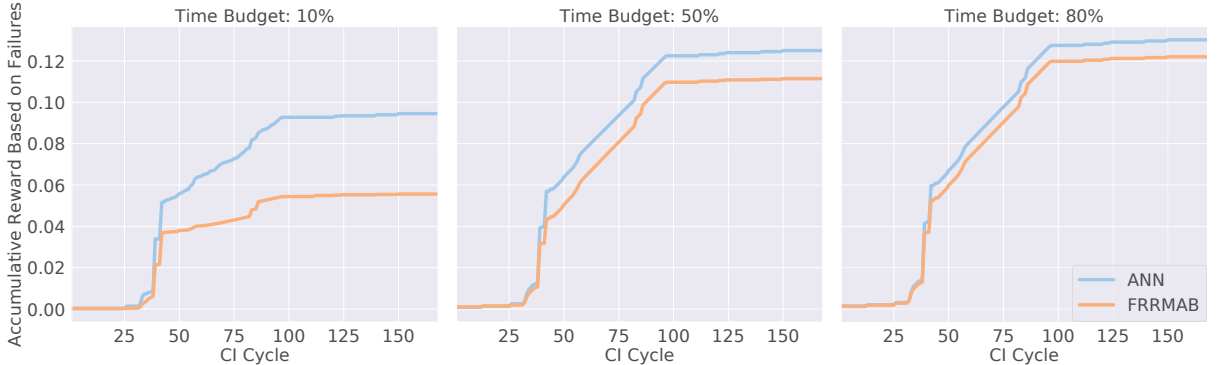
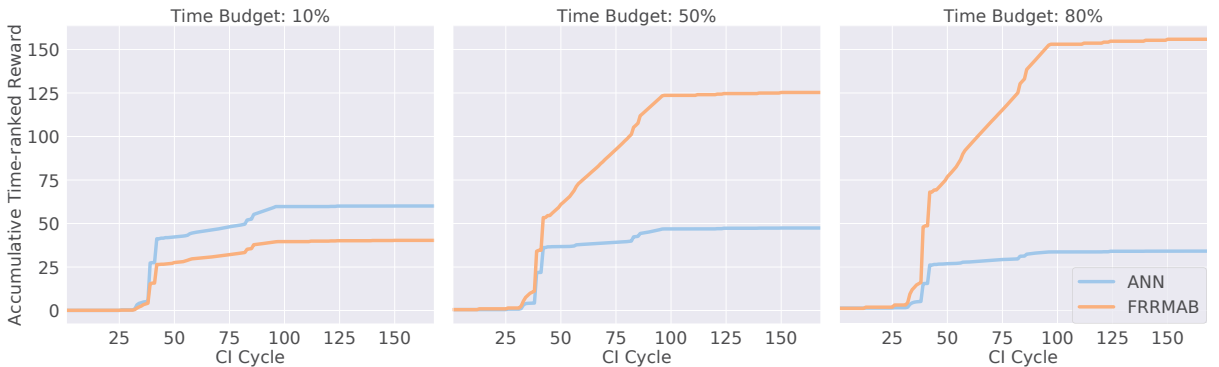
However, we need to consider the impact of this time in the complete *CI* cycle and the interval between the *CI* cycles, and consequently, to observe whether the spent prioritization time is expensive. In addition, it is important to analyze the impact concerning the time spent to reveal the first failure, which can be used to reduce the time spent in a *CI* cycle once that the test execution cost can be reduced when a failure is revealed because the test is ended. For this end, Table 4.5 presents the time reduction average, given by the indicator *NTR* (Equation 4.3) with both reward functions, over the three budgets. Refer to Table 4.2 to see duration of the *CI* cycles and the interval between them.

Table 4.5 shows percentages of time reduction provided by our approach are close to 73% for *IOF/ROL*, 47% for *Deeplearning4j*, and 43% for *Druid*. This reduction in a *CI* cycle is due to the early fault detection provided by our approach. But, in most systems, we can see a low percentage. The performance of both functions is similar, as well as the *NTR* values for each system. In overall, the *RNFail* function obtains better reduction than *TimeRank* considering time

Table 4.5: Mean and standard deviation *NAPFD*, *APFD_c*, and *NTR* values: *COLEMAN* against *RETECS*.

(See caption of Table 4.3 for a description of the headings.)

SUT	NAPFD				APFD _c				NTR			
	RNFail		TimeRank		RNFail		TimeRank		RNFail		TimeRank	
	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB
Time Budget: 10%												
Druid	0.6768 ± 0.129 ▲	0.6801 ± 0.052 ★	0.6488 ± 0.074 ▲	0.7137 ± 0.074 ★	0.8067 ± 0.088 ★	0.6964 ± 0.063 ▲	0.7815 ± 0.051 ★	0.7181 ± 0.076 △	0.1863 ± 0.124	0.2027 ± 0.115	0.1556 ± 0.077	0.2355 ± 0.135
Fastjson	0.8713 ± 0.015 ▲	0.9030 ± 0.014 ★	0.8719 ± 0.005 ▲	0.8980 ± 0.018 ★	0.8805 ± 0.014 ▲	0.9064 ± 0.013 ★	0.8810 ± 0.004 ▲	0.8974 ± 0.018 ★	0.0194 ± 0.016	0.3117 ± 0.221	0.0219 ± 0.007	0.2674 ± 0.232
DeepLearning4j	0.6615 ± 0.072 ▲	0.7533 ± 0.002 ★	0.6739 ± 0.016 ▲	0.7716 ± 0.000 ★	0.8135 ± 0.023 ★	0.7766 ± 0.001 ▲	0.8185 ± 0.010 ★	0.7773 ± 0.000 ▲	0.5488 ± 0.029	0.4626 ± 0.001	0.5546 ± 0.015	0.4663 ± 0.000
DBSpace	0.9437 ± 0.001 ▲	0.9489 ± 0.003 ★	0.9410 ± 0.001 ▲	0.9496 ± 0.004 ★	0.9480 ± 0.001 ▲	0.9510 ± 0.002 ★	0.9458 ± 0.001 ▲	0.9513 ± 0.004 ★	0.0188 ± 0.001	0.0370 ± 0.015	0.0105 ± 0.001	0.0393 ± 0.018
Guava	0.9676 ± 0.015 ★	0.9554 ± 0.002 ▲	0.9563 ± 0.004	0.9586 ± 0.001 ★	0.9811 ± 0.007 ★	0.9561 ± 0.001 ▲	0.9761 ± 0.003 ★	0.9582 ± 0.001 ▲	0.0449 ± 0.015	0.0471 ± 0.016	0.0367 ± 0.005	0.0492 ± 0.015
OkHttp	0.8357 ± 0.002 ★	0.8323 ± 0.000 ▲	0.8095 ± 0.006 ▲	0.8407 ± 0.000 ★	0.8484 ± 0.001 ★	0.8378 ± 0.000 ▲	0.8292 ± 0.006 ▲	0.8425 ± 0.000 ★	0.0830 ± 0.001	0.0658 ± 0.000	0.0579 ± 0.008	0.0702 ± 0.000
Retrofit	0.9641 ± 0.001 ★	0.9639 ± 0.000	0.9621 ± 0.001 ▲	0.9642 ± 0.000 ★	0.9672 ± 0.001 ★	0.9646 ± 0.000 ▲	0.9655 ± 0.001 ★	0.9648 ± 0.000 △	0.0088 ± 0.000	0.0070 ± 0.000	0.0076 ± 0.001	0.0073 ± 0.000
Zxing	0.9854 ± 0.000 ★	0.9826 ± 0.000 ▲	0.9855 ± 0.000 ★	0.9828 ± 0.000 ▲	0.9893 ± 0.000 ★	0.9832 ± 0.000 ▲	0.9893 ± 0.000 ★	0.9835 ± 0.000 ▲	0.0122 ± 0.000	0.0037 ± 0.000	0.0126 ± 0.001	0.0037 ± 0.000
IOF/RDL	0.3704 ± 0.005 ★	0.3632 ± 0.001 ▲	0.3779 ± 0.003 ★	0.3670 ± 0.001 ▲	0.3746 ± 0.005 ★	0.3661 ± 0.001 ▲	0.3819 ± 0.003 ★	0.3701 ± 0.001 ▲	0.5133 ± 0.030	0.5585 ± 0.007	0.5646 ± 0.015	0.5701 ± 0.004
Paint Control	0.9078 ± 0.000 ★	0.9076 ± 0.000 ▲	0.9077 ± 0.000 ★	0.9076 ± 0.000 ▲	0.9082 ± 0.000 ★	0.9080 ± 0.000 ▲	0.9081 ± 0.000 ★	0.9080 ± 0.000 ▲	0.1151 ± 0.000	0.1133 ± 0.000	0.1139 ± 0.000	0.1131 ± 0.000
GSDTSR	0.9893 ± 0.000 ★	0.9894 ± 0.000 ★	0.9893 ± 0.000 ★	0.9894 ± 0.000 ★	0.9894 ± 0.000 ★	0.9894 ± 0.000 ★	0.9894 ± 0.000 ★	0.9894 ± 0.000 ★	0.0087 ± 0.000	0.0093 ± 0.000	0.0090 ± 0.000	0.0096 ± 0.000
Average	0.8340	0.8436	0.8294	0.8494	0.8643	0.8487	0.8294	0.8494	0.1428	0.1653	0.1404	0.1665
Time Budget: 50%												
Druid	0.6851 ± 0.134 ▲	0.9333 ± 0.013 ★	0.6323 ± 0.074 ▲	0.9710 ± 0.008 ★	0.8147 ± 0.102 ▲	0.9486 ± 0.016 ★	0.7597 ± 0.051 ▲	0.9787 ± 0.009 ★	0.1840 ± 0.137	0.4057 ± 0.013	0.1213 ± 0.071	0.4225 ± 0.008
Fastjson	0.8714 ± 0.007 ▲	0.9174 ± 0.021 ★	0.8902 ± 0.013 ▲	0.9118 ± 0.028 ★	0.9326 ± 0.005 ★	0.9186 ± 0.021 ▲	0.9392 ± 0.017 ★	0.9140 ± 0.027 ▲	0.0399 ± 0.010	0.3539 ± 0.262	0.0602 ± 0.020	0.3394 ± 0.282
DeepLearning4j	0.7049 ± 0.070 ▲	0.7890 ± 0.001 ★	0.6562 ± 0.018 ▲	0.8200 ± 0.001 ★	0.8331 ± 0.041 ★	0.8106 ± 0.001 △	0.8379 ± 0.012 ★	0.8134 ± 0.001 ▲	0.5276 ± 0.039	0.4695 ± 0.000	0.5447 ± 0.010	0.4625 ± 0.000
DBSpace	0.9568 ± 0.001 ▲	0.9724 ± 0.009 ★	0.9485 ± 0.001 ▲	0.9766 ± 0.005 ★	0.9683 ± 0.001 ▲	0.9737 ± 0.009 ★	0.9615 ± 0.001 ▲	0.9767 ± 0.009 ★	0.0334 ± 0.001	0.0751 ± 0.031	0.0218 ± 0.002	0.0776 ± 0.034
Guava	0.9502 ± 0.015 ▲	0.9653 ± 0.004 ★	0.9578 ± 0.004 ▲	0.9675 ± 0.007 ★	0.9767 ± 0.008 ★	0.9687 ± 0.003 ▲	0.9806 ± 0.005 ★	0.9672 ± 0.007 ▲	0.0303 ± 0.016	0.0662 ± 0.024	0.0387 ± 0.006	0.0659 ± 0.020
OkHttp	0.8812 ± 0.010 ▲	0.9192 ± 0.000 ★	0.8446 ± 0.003 ▲	0.9317 ± 0.000 ★	0.8878 ± 0.015 ▲	0.9177 ± 0.000 ★	0.8869 ± 0.002 ▲	0.9246 ± 0.000 ★	0.1118 ± 0.014	0.1431 ± 0.000	0.1060 ± 0.003	0.1486 ± 0.000
Retrofit	0.9706 ± 0.002 ▲	0.9853 ± 0.000 ★	0.9718 ± 0.002 ▲	0.9893 ± 0.000 ★	0.9762 ± 0.002 ▲	0.9850 ± 0.000 ★	0.9778 ± 0.002 ▲	0.9885 ± 0.000 ★	0.0134 ± 0.001	0.0156 ± 0.000	0.0138 ± 0.003	0.0172 ± 0.000
Zxing	0.9878 ± 0.000 ★	0.9846 ± 0.000 ▲	0.9881 ± 0.001 ★	0.9857 ± 0.000 ▲	0.9954 ± 0.000 ★	0.9862 ± 0.000 ▲	0.9956 ± 0.001 ★	0.9869 ± 0.000 ▲	0.0201 ± 0.000	0.0109 ± 0.000	0.0201 ± 0.001	0.0110 ± 0.000
IOF/RDL	0.5101 ± 0.007 ★	0.5046 ± 0.002 ▲	0.5025 ± 0.006 ▲	0.5189 ± 0.002 ★	0.5175 ± 0.008 ★	0.5081 ± 0.002 ▲	0.5043 ± 0.006 ▲	0.5223 ± 0.002 ★	0.7037 ± 0.019	0.7110 ± 0.003	0.6834 ± 0.014	0.7193 ± 0.003
Paint Control	0.9150 ± 0.000 ★	0.9150 ± 0.000 ▲	0.9138 ± 0.000 ▲	0.9150 ± 0.000 ★	0.9171 ± 0.000 ★	0.9162 ± 0.000 ▲	0.9140 ± 0.000 ▲	0.9162 ± 0.000 ★	0.1283 ± 0.000	0.1222 ± 0.000	0.1142 ± 0.001	0.1223 ± 0.000
GSDTSR	0.9911 ± 0.000 ★	0.9893 ± 0.000 ★	0.9893 ± 0.000 ★	0.9894 ± 0.000 ★	0.9911 ± 0.000 ★	0.9894 ± 0.000 ★	0.9910 ± 0.000 ★	0.9894 ± 0.000 ★	0.0199 ± 0.000	0.0093 ± 0.000	0.0179 ± 0.000	0.0096 ± 0.000
Average	0.8567	0.8978	0.8451	0.9070	0.8919	0.9021	0.8862	0.9071	0.1648	0.2166	0.1583	0.2178
Time Budget: 80%												
Druid	0.6490 ± 0.113 ▲	0.9380 ± 0.012 ★	0.6551 ± 0.099 ▲	0.9830 ± 0.003 ★	0.7142 ± 0.111 ▲	0.9469 ± 0.015 ★	0.6881 ± 0.090 ▲	0.9912 ± 0.004 ★	0.1477 ± 0.113	0.4069 ± 0.014	0.1230 ± 0.096	0.4292 ± 0.004
Fastjson	0.8708 ± 0.007 ▲	0.9536 ± 0.010 ★	0.8925 ± 0.010 ▲	0.9242 ± 0.028 ★	0.9037 ± 0.008 ▲	0.9488 ± 0.012 ★	0.9133 ± 0.015	0.927 ± 0.026 ★	0.0385 ± 0.011	0.4613 ± 0.278	0.0516 ± 0.018	0.3963 ± 0.301
DeepLearning4j	0.7058 ± 0.091 ▲	0.8424 ± 0.001 ★	0.6640 ± 0.016 ▲	0.8641 ± 0.001 ★	0.8158 ± 0.064 ★	0.8068 ± 0.002 ▽	0.8522 ± 0.012 ★	0.7989 ± 0.001 ▲	0.5016 ± 0.058	0.4224 ± 0.001	0.5475 ± 0.007	0.4047 ± 0.000
DBSpace	0.9601 ± 0.001 ▲	0.9792 ± 0.006 ★	0.9508 ± 0.001 ▲	0.9825 ± 0.007 ★	0.9738 ± 0.001 △	0.9796 ± 0.006 ★	0.9639 ± 0.001 ▲	0.9810 ± 0.008 ★	0.0374 ± 0.001	0.0800 ± 0.031	0.0269 ± 0.002	0.0812 ± 0.034
Guava	0.9441 ± 0.012 ▲	0.9784 ± 0.012 ★	0.9581 ± 0.007 ▲	0.9841 ± 0.014 ★	0.9627 ± 0.010 ▲	0.9780 ± 0.013 ★	0.9689 ± 0.009 △	0.9825 ± 0.015 ★	0.0247 ± 0.013	0.0812 ± 0.044	0.0348 ± 0.009	0.0888 ± 0.045
OkHttp	0.9027 ± 0.013 ▲	0.9350 ± 0.000 ★	0.8358 ± 0.004 ▲	0.9478 ± 0.000 ★	0.8836 ± 0.020 ▲	0.9271 ± 0.000 ★	0.8874 ± 0.003 ▲	0.9362 ± 0.000 ★	0.1112 ± 0.017	0.1493 ± 0.000	0.1153 ± 0.003	0.1551 ± 0.000
Retrofit	0.9724 ± 0.005 ▲	0.9881 ± 0.000 ★	0.9745 ± 0.003 ▲	0.9916 ± 0.000 ★	0.9785 ± 0.004 ▲	0.9873 ± 0.000 ★	0.9808 ± 0.003 ▲	0.9903 ± 0.000 ★	0.0140 ± 0.001	0.0161 ± 0.000	0.0145 ± 0.001	0.0179 ± 0.000
Zxing	0.9878 ± 0.000 ▲	0.9972 ± 0.000 ★	0.9883 ± 0.001 ▲	0.9996 ± 0.000 ★	0.9953 ± 0.000 ▲	0.9975 ± 0.000 ★	0.9953 ± 0.001 ▲	0.9996 ± 0.000 ★	0.0201 ± 0.000	0.0224 ± 0.000	0.0197 ± 0.002	0.0227 ± 0.000
IOF/RDL	0.5495 ± 0.006 ▲	0.5569 ± 0.002 ★	0.5287 ± 0.007 ▲	0.5678 ± 0.001 ★	0.5593 ± 0.006 ★	0.5591 ± 0.002	0.5311 ± 0.006 ▲	0.5699 ± 0.001 ★	0.7263 ± 0.015	0.7293 ± 0.002	0.6857 ± 0.011	0.7334 ± 0.001
Paint Control	0.9162 ± 0.000 ▲	0.9171 ± 0.000 ★	0.9160 ± 0.000 ▲	0.9171 ± 0.000 ★	0.9176 ± 0.000 ★	0.9176 ± 0.000 ▲	0.9158 ± 0.000 ▲	0.9177 ± 0.000 ★	0.1209 ± 0.000	0.1285 ± 0.000	0.1161 ± 0.001	0.1204 ± 0.001
GSDTSR	0.9921 ± 0.000 ★	0.9893 ± 0.000 ★	0.9914 ± 0.000 ★	0.9894 ± 0.000 ★	0.9919 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9917 ± 0.000 ★	0.9894 ± 0.000 ★	0.0218 ± 0.001	0.0093 ± 0.000	0.0203 ± 0.000	0.0096 ± 0.000
Average	0.8591	0.9159	0.8523	0.9228	0.8816	0.9126	0.8817	0.9167	0.1611	0.2272	0.1596	0.2236

Figure 4.10: Reward Based on Failures (*RNFail*) function.Figure 4.11: Time-Ranked Reward (*TimeRank*) function.Figure 4.12: Accumulative Reward values over the *CI* cycles from Druid system.

budget of 80%, and the *TimeRank* is better in the other time budgets. Comparing the average percentage for the budget of 50% we observe an improvement regarding the budget of 10%. But

Table 4.6: Mean and standard deviation *RFTC* values: *COLEMAN* against *RETECS*.

SUT	RFTC			
	RNFail		TimeRank	
	ANN	FRRMAB	ANN	FRRMAB
TIME BUDGET: 10%				
Druid	1166.4411 ± 546.049 ▲	209.3289 ± 98.987 ★	1240.8713 ± 356.817 ▲	56.1579 ± 31.056 ★
Fastjson	1094.1147 ± 209.106 ▲	184.9911 ± 65.066 ★	998.7949 ± 177.738 ▲	96.0378 ± 36.22 ★
Deeplearning4j	5.3151 ± 2.456 ▲	2.3049 ± 0.048 ★	5.7919 ± 0.677 ▲	2.0437 ± 0.007 ★
DSpace	11.9561 ± 0.753 ▲	3.024 ± 1.33 ★	13.6276 ± 0.754 ▲	2.1428 ± 0.67 ★
Guava	129.3477 ± 99.443 ▲	31.8991 ± 20.546 ★	191.1219 ± 29.545 ▲	15.0977 ± 13.75 ★
OkHttp	7.4112 ± 0.397 ▲	4.3424 ± 0.0 ★	15.8935 ± 1.951 ▲	1.8039 ± 0.0 ★
Retrofit	3.7352 ± 0.439 ▲	1.5152 ± 0.0 ★	4.7297 ± 0.772 ▲	1.7941 ± 0.0 ★
ZXing	39.7929 ± 0.643 ▲	1.0 ± 0.0 ★	39.1204 ± 1.701 ▲	1.0 ± 0.0 ★
IOF/ROL	1.6445 ± 0.158 ▲	1.2626 ± 0.064 ★	1.5639 ± 0.108 ▲	1.0992 ± 0.05 ★
Paint Control	1 ± 0.0	1 ± 0.0	1.001 ± 0.002 ▽	1.0 ± 0.0 ★
GSDTSR	3.2553 ± 0.387 ▲	2.1316 ± 0.1 ★	3.8958 ± 0.18 ▲	1.1482 ± 0.071 ★
TIME BUDGET: 50%				
Druid	1225.6197 ± 600.746 ▲	121.8396 ± 43.763 ★	1420.3143 ± 418.926 ▲	51.2697 ± 11.926 ★
Fastjson	1527.9434 ± 93.004 ▲	315.8923 ± 79.836 ★	1173.9871 ± 321.789 ▲	335.9929 ± 125.629 ★
Deeplearning4j	5.0267 ± 1.827 ▲	2.5496 ± 0.011 ★	7.3264 ± 0.530 ▲	2.464 ± 0.005 ★
DSpace	19.0354 ± 0.887 ▲	5.5312 ± 1.939 ★	27.6301 ± 0.921 ▲	4.1089 ± 1.769 ★
Guava	289.1586 ± 99.054 ▲	78.6409 ± 45.928 ★	235.0919 ± 27.865 ▲	24.0869 ± 21.479 ★
OkHttp	6.203 ± 2.066 ▲	4.188 ± 0.014 ★	19.6306 ± 0.79 ▲	2.3643 ± 0.002 ★
Retrofit	5.4302 ± 0.43 ▲	2.4059 ± 0.0 ★	5.625 ± 0.415 ▲	1.4299 ± 0.0 ★
ZXing	51.2576 ± 0.579 ▲	5.6 ± 0.0 ★	49.4232 ± 3.15 ▲	2.0 ± 0.0 ★
IOF/ROL	1.8009 ± 0.292 ▲	1.2588 ± 0.035 ★	1.9213 ± 0.218 ▲	1.151 ± 0.024 ★
Paint Control	1.0234 ± 0.004 ▲	1.0018 ± 0.001 ★	1.0257 ± 0.007 ▲	1.0014 ± 0.001 ★
GSDTSR	1.9072 ± 0.06 ★	2.1461 ± 0.101 ▲	3.5648 ± 0.141 ▲	1.1505 ± 0.072 ★
TIME BUDGET: 80%				
Druid	1427.8103 ± 493.429 ▲	146.9112 ± 46.436 ★	1035.3369 ± 658.042 ▲	50.3805 ± 11.25 ★
Fastjson	1535.2998 ± 101.625 ▲	398.4345 ± 105.27 ★	993.382 ± 393.441 ▲	572.0954 ± 221.573 ★
Deeplearning4j	5.5748 ± 2.358 ▲	2.8146 ± 0.014 ★	7.8533 ± 0.581 ▲	2.5017 ± 0.011 ★
DSpace	23.126 ± 1.021 ▲	6.8651 ± 2.946 ★	33.4617 ± 1.119 ▲	6.0794 ± 3.343 ★
Guava	330.5342 ± 74.0 ▲	84.6856 ± 34.075 ★	202.2301 ± 47.258 ▲	83.9989 ± 78.446 ★
OkHttp	4.2988 ± 2.242	4.0748 ± 0.021	21.5784 ± 1.148 ▲	2.282 ± 0.003 ★
Retrofit	5.9252 ± 0.884 ▲	2.4636 ± 0.0 ★	5.8832 ± 0.587 ▲	1.4386 ± 0.0 ★
ZXing	51.0061 ± 1.233 ▲	4.2727 ± 0.0 ★	48.6848 ± 2.926 ▲	1.3636 ± 0.0 ★
IOF/ROL	2.021 ± 0.447 ▲	1.317 ± 0.026 ★	2.5248 ± 0.362 ▲	1.2366 ± 0.017 ★
Paint Control	1.015 ± 0.002 ▲	1.0003 ± 0.001 ★	1.0344 ± 0.009 ▲	1.0003 ± 0.001 ★
GSDTSR	1.9413 ± 0.322 ★	2.1461 ± 0.101 ▲	3.4858 ± 0.112 ▲	1.1505 ± 0.072 ★

this difference is slightly lower when we compare the budgets of 50% and 80%. Then, the budget may be a reason to explain low reduction. Other possible reasons are the difficulty inherent to the *TCPCI* problem and the need to consider other aspects in the prioritization, such as the time to execute each test case.

Regarding the time spent in a *CI* cycle and interval between commits for each system, we observe that a new commit is typically performed after a *CI* cycle is finished and with a considered time, due to the time between commits is, in most systems, higher than the time spent by a *CI* cycle. In this way, the systems chosen do not present a situation with multiple test requests. In addition, the time spent in a *CI* cycle and between cycles is in minutes, and as our approach spent, in the worst case, less than one second to prioritize the test cases, there is not a negative impact in the use of our approach.

RQ2: *Our approach is applicable to the CI context. It contributes to reducing the time spent in a CI cycle, even with a restrictive budget of 10%. In some cases, the reduction can achieve a percentage of 70%.*

4.6.3 RQ3: Comparing *COLEMAN* and *RETECS*

To answer RQ3, we compare *COLEMAN* using *FRRMAB* (the best policy according to RQ1) with *RETECS* using *ANN*. Table 4.5 shows the *NAPFD* and *APFD_c* results obtained.

As we can observe for *RNFail* function and *NAPFD*, *FRRMAB* stands out in $\approx 70\%$ of the cases (23 out of 33). *ANN* stands out in 33% (11 cases). *FRRMAB* is the best with statistical difference in $\approx 66\%$ (22 out of 33) of the cases against 30% (10 cases) for *ANN*. Regarding the *NAPFD* average across the *SUTs*, *FRRMAB* obtained the best results, with statistical difference in the time budget of 80% . With the *TimeRank* function, *FRRMAB* presents even better results. *FRRMAB* stands out in $\approx 82\%$ (27 out of 33) of the cases against 21% (7 cases) for *ANN*. *FRRMAB* is the best with statistical difference in $\approx 79\%$ of the cases (26 out of 33) against 18% (6 cases) for *ANN*. Across the *SUTs*, *FRRMAB* is the best, with statistical difference considering the time budgets of 50% and 80% .

We observe that *ANN* has a bad performance for *Druid* and *DeepLearning4j*, and improves performance in comparison with *FRRMAB* when the budget decreases. On the other hand, *FRRMAB* presents more stability (low variation) in *NAPFD* average values than *ANN*. *FRRMAB* has the best performance with low variation and better values for *Druid* and *DeepLearning4j*. Regarding *APFDc* values, *ANN* using *RNFail* function stands out in 64% of the cases (21 out of 33) against 45% (15 cases) for *FRRMAB*. *ANN* is the best with statistical difference in $\approx 55\%$ (18 out of 33) of the cases against 30% (10) for *FRRMAB*. Considering *TimeRank* the opposite occurs. *FRRMAB* stands out in 55% of the cases (18 out of 33), against 48% (16) for *ANN*. *FRRMAB* is the best with statistical difference in $\approx 52\%$ (17 out of 33) of the cases against 42% (14) for *ANN*. Across the *SUTs*, overall, *FRRMAB* and *ANN* are statistically equivalent in most of the time budgets and reward functions, except in *TimeRank* function with a time budget of 80% in which *FRRMAB* is better than *ANN*. For this indicator, the results show the approaches are competitive, that is, there is not a great difference between them. A possible reason for this is that *RETECS* considers the individual test case duration, and *COLEMAN*, differently, considers only historical failure data.

FRRMAB with *TimeRank* produces better values concerning *NAPFD*, *APFDc*, and time budgets of 50% and 80% . But overall, the results of both functions are similar.

According to Table 4.4, *ANN* is a bit faster than *FRRMAB* considering the function *RNFail*, and slower considering *TimeRank*. *FRRMAB* is stable for both functions, whilst *ANN* spends more time using *TimeRank* function. Analyzing the prioritization time spent along with *CI* cycles, we observe that both approaches spend more time when test cases are added or removed from one cycle to another. This occurs because both approaches need to update the information about the test cases, mainly when a high number of test cases increases or decreases abruptly. But a greater impact is observed for *RETECS*. In particular, *ANN* has a great variation in the spent time for *IOF/ROL*. For this system, *RETECS* can take in the worst case more than one second.

Regarding *NTR*, for the three budgets, *FRRMAB* is the best in $\approx 64\%$ (21 out of 33) of the cases against 36% (12) for *ANN* for *RNFail* function. Considering *TimeRank*, *FRRMAB* is the best in $\approx 73\%$ (24) of the cases against $\approx 27\%$ (9) for *ANN*. Regarding *RFTC* (Table 4.6) and *RNFail*, *FRRMAB* stands out in $\approx 94\%$ of the cases (31 out of 33) against $\approx 12\%$ (4) for *ANN*. But in 2 of these 4 cases, *ANN* and *FRRMAB* are statistically equivalent. With *TimeRank*, *FRRMAB* is the best in all cases.

With the systems used, we do not identify a scalability pattern concerning the time spent to prioritize the test cases. The use of systems with a great number of test cases in each *CI* cycles can help in the identification of our approach scalability in future studies.

Due to the bad performance of *ANN* in *Druid*, we analyze this system. *Druid* has the lowest number of *CI* cycles (168) among the systems evaluated, as well as it has many tests (2391). Figure 4.12 shows the accumulative reward values from *ANN* (blue line) and *FRRMAB* (orange line) along the *CI* cycles considering *RNFail* and *TimeRank* functions. We can observe that although *RETECS* better assigns the rewards in the *RNFail* function, the prioritization order

is not adequate, which we can see with the *TimeRank* function. In this way, *FRRMAB* better mitigates than *ANN* the problem of beginning without learning, mainly in combination with the *TimeRank* function, and adapts quickly to deal with a peak of faults in the first cycles⁸.

RQ3: *COLEMAN* using *FRRMAB* outperforms *RETECS* using *ANN* regarding *NAPFD*, *RFTC* and *NTR* indicators, independently of the reward functions and budgets investigated. Regarding *APFDc*, both approaches present similar results. Besides, with respect to the execution time, *FRRMAB* is more stable, that is, adapts better to deal with peak of faults.

4.7 CONCLUDING REMARKS

This chapter presented *COLEMAN*, an approach to deal with the *TCPCI* problem. *COLEMAN* is based on *MAB* with combinatorial and volatile characteristics to dynamically obtain an adequate prioritized test suite for each *CI* cycle (commit) using historical failure data of test cases. The approach properly deals with the *EvE* dilemma and takes into account *TCPCI* characteristics such as test case volatility.

We evaluated our approach concerning three time budgets: 10%, 50%, and 80%; with five *MAB* policies: *FRRMAB*, *UCB*, ϵ -Greedy, Greedy, and Random; and compared it against an RL-based approach from literature, named *RETECS* [90]. The results show that *FRRMAB* policy outperforms the other policies with both reward functions. In most cases, our approach outperforms *RETECS* in terms of *NAPFD*, *NTR*, and earlier fault detection (*RFTC*). It also does not present great variations in the prioritization time, and does not require any additional information. Regarding *APFDc*, the obtained values are competitive with *RETECS*.

Our approach spends, in the worst case, less than one second to prioritize a test suite, and the obtained results in many aspects investigated show high performance. Then, we can conclude that *COLEMAN* contributes efficiently and effectively to address the *TCPCI* problem.

Due to the promising results observed in this chapter, we explored *COLEMAN* against other approaches, such as a search-based approach (Appendix C) and a deterministic one (Appendix D), as well as in the *TCPCI* context for *HCS* (Appendices E and F). In the next chapter, we provide a summary of the results presented in these papers.

⁸See supplementary material about characteristics of this system.

5 OTHER EVALUATIONS

This chapter presents a summary of the results obtained in the other evaluations conducted with *COLEMAN*. Section 5.1 presents an evaluation against a search-based approach, and Section 5.2 against to a deterministic one. Section 5.3 presents the results from two studies conducted in the *HCS* context. We derived Table 5.1 containing for each study a summary of the approaches, evaluation metrics, and reward functions used.

Table 5.1: Summary of the main characteristics of the results obtained in the other evaluations.

This table presents the approaches, evaluation metrics, and reward functions used in the papers published. The first column contains the reference of the study, followed by whether the publication is in the HCS testing. Columns 3 to 7 show the *Approaches Under Evaluation* of each study, where “COL.” means *COLEMAN*, “RET.” *RETECS*, “Ran.” *Random*, “GA” *Genetic Algorithm*, and “Det.” *Deterministic*. Columns 8 to 13 show the *Evaluation Metric* used for prioritization assessment, where “NAPFD” means *Normalized Average Percentage of Faults Detected*, “APFDc” *Average Percentage of Faults Detected with cost consideration*, “RFTC” *Rank of the Failing Test Cases*, “NTR” *Normalized Time Reduction*, “PR” *Prioritization Time*, and “RMSE” *Root-Mean-Square Error*. Columns 14 to 15 show the used *Reward Function* (see Section 4.3.1). The last column shows the number of systems evaluated and, if available, the number of variants in parenthesis.

Ref.	HCS	Approaches Under Evaluation					Evaluation Metric						Reward Function		N# Systems (Variants)
		COL.	RET.	Ran.	GA	Det.	NAPFD	APFDc	RFTC	NTR	PR	RMSE	RNFail	TimeRank	
[72]	-	✓	-	-	✓	-	✓	-	-	-	✓	✓	✓	✓	7 (0)
[77]	-	✓	✓	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	12 (0)
[64]	✓	✓	-	✓	-	-	✓	-	✓	✓	✓	✓	-	✓	1 (31)
[65]	✓	✓	✓	✓	-	✓	✓	-	-	✓	✓	✓	-	✓	2 (63)

5.1 COMPARISON WITH A SEARCH-BASED APPROACH

Although search-based algorithms are not suitable to the *TCPCI* context because it takes too long to execute, this kind of approach can find near-optimal solutions. Considering this fact, we analysed the trade-offs of the *COLEMAN* solutions in comparison with the near-optimal solutions generated by a *GA* [72]. The full text is available in Appendix C.

In this study, for the *GA* algorithm, we represent each individual in the population as a possible prioritization for the test case available in a commit. The fitness function is based on the *NAPFD* metric. Besides that, the test results are already known by the algorithm. Then, we observe the *TCPCI* problem’s difficulty in obtaining a feasible solution from the search space.

Regarding the evaluation process, we used seven large-scale real-world software systems and three different test budgets, considering that the time available for a *CI* cycle corresponds to respectively 10%, 50% and 80% of the total time required to execute the test case set available for that cycle. We used three measures that better fit with time constraints: *NAPFD*, *Root-Mean-Square Error (RMSE)*, and *Prioritization Time*. *RMSE* metric was introduced in this study and allows us to observe the distance between the solutions provided by *COLEMAN* and *GA*, that is, it shows the accuracy of an algorithm.

The outcomes show that *COLEMAN* provides solutions that are near (or very near) to the *GA* solutions in $\approx 90\%$ of the cases, considering all systems, budgets, and both reward functions. Concerning the prioritization time, the approach spends less than one second in the worst case, even in challenging systems for the *TCPCI* problem, while *GA* spends in the worst

case around 31 seconds. This allows us to observe how complex the search space of the *TCPCI* problem is, in which even knowing the test results in advance, *GA* takes a considerable time to find reasonable solutions.

In this way, the main contribution of this paper is to conduct a trade-off analysis of our online learning approach *COLEMAN*. The analysis points out research directions for *TCPCI* considering the drawbacks and strengths found. Besides that, we provide supplementary material with the source code of the Genetic Algorithm, additional analysis, and results. This replication package allows future investigations [73].

5.2 COMPARISON WITH A DETERMINISTIC APPROACH

In contrast with the study presented in Section 5.1, we compared the solutions produced by two approaches considered as state-of-the-art in *TCPCI*, *COLEMAN* and *RETECS*, against the solutions produced by a deterministic approach [77]. The evaluation aimed to observe how far the solutions produced by these approaches are from optimal solutions, as well as to provide a guideline for future research, benchmark, and to present the limitations and room for improvements. In addition to this, we made available a replication package containing supplementary material [78]. The full text is available in Appendix D.

The conducted evaluation considered six evaluation measures to observe the results in different perspectives: fault detection effectiveness (*NAPFD*), fault detection effectiveness with cost consideration (*APFDc*), early fault detection (*RFTC*), test time reduction (*NTR*), prioritization time, and accuracy (*RMSE*). For this evaluation, we considered 12 real-world software systems, three time budgets (10%, 50%, and 80%), and two reward functions (*RNFail* and *TimeRank*).

Considering the *NAPFD* values, we observed that *COLEMAN* presents better performance than *RETECS*. Using *TimeRank*, *COLEMAN* obtained results that are equivalent to optimal in 42% of the cases. This percentage increases in the presence of less restrictive budgets, reaching 58% of the cases in the time budget of 80%. Besides that, the learning-based approaches have the worst performance in systems with a high test case volatility and a few number of *CI* cycles. One important finding is related to the failure distribution, in which a high number of failures distributed over many test cases makes the *TCP* task harder.

Regarding *APFDc* values, *COLEMAN* outperforms *RETECS* in most cases. However, *RETECS* has a better performance in the most restrictive budget of 10%. The analysis of *APFDc* using test case duration as cost leads to results similar to those obtained in the *NAPFD* analysis. In general, the *APFDc* values are close to the optimal, and the more significant differences are obtained to the same systems.

Considering *RFTC* and *NTR* values, *COLEMAN* using *TimeRank* obtained performance equivalent to the optimal results in $\approx 70\%$ of the cases. Moreover, *COLEMAN* is better than *RETECS* in all cases. The *RFTC* and *NTR* metrics provided an interesting finding. Even using a deterministic approach, sometimes, the test time reduction is low due to peaks of failures, failure distributed across the test cases, and variation of the failing test cases over *CI* cycles. Nevertheless, *COLEMAN* reached high percentages of reductions for systems, considered hard cases for prioritizing.

Although the learning-based approaches produced good results, the prioritization time show us whether they are applicable in real scenarios. Overall, they spend less than one second to execute. Consequently, both approaches are applicable.

The last analysis considered the *RMSE* values. In this analysis, we observed that a high test case volatility combined with an increasing number of failures may be a limitation for

RETECS. Based on the *RMSE* values found, we defined an scale. Based on the proposed scale, we observed that *COLEMAN* finds reasonable solutions in 92% of the cases and *RETECS* in 75%. The less restrictive the budget the greater the number of reasonable solutions found by both approaches. We can then conclude that the solutions generated are very close to the optimal ones.

Considering the facts aforementioned, *COLEMAN* outperformed *RETECS* in the great majority of the cases, considering all systems, budgets, and measures. Furthermore, both approaches are applicable in real scenarios and spending negligible time to execute. Both approaches allow reducing the *CI* cycle's time cost.

5.3 APPLICATION FOR HCS

HCS testing is usually costly, as a significant number of variants need to be tested. This becomes more problematic when *CI* practices are adopted. To address *CI* challenges, *COLEMAN* has been successfully applied. However, our approach does not consider *HCS* particularities such as the volatility of variants, in which some variants can be included/discontinued along with *CI* cycles. In order to allow the applicability of our approach in the *HCS* context, as well as for learning-based approaches designed for *TCPCI*, we conducted two studies [64, 65] (the second study [65] is an extension). The full texts are available in Appendices E and F. We made available the replication packages containing supplementary materials from both studies [66, 67].

In such studies, we introduced two strategies for applying learning-based approaches for TCP in the *CI* of *HCS*: the *VTS* that relies on the test set specific for each variant; and the *WTS* that prioritizes the test set composed by the union of the test cases of all variants.

Both strategies were applied to two real-world *HCS*s, considering three test budgets (10%, 50%, and 80%), and evaluated regarding some indicators such as early fault detection, time reduction, and accuracy. We used four approaches in the evaluation: *COLEMAN*, *RETECS*, Random, and a deterministic approach. The main idea is to determine how far the solutions produced by the strategies using each one of the approaches are from the optimal solutions produced by the deterministic approach.

The outcomes show the strategies using *COLEMAN* are very cost-effective and better than the other approaches, in which it produces more than 92% of the cases reasonable solutions that are near to the optimal solutions obtained by a deterministic approach. Regarding the prioritization time, both strategies spend less than one second. *WTS* provides better results in the less restrictive budgets, and *VTS* the opposite. The main advantage of *WTS* is that if a new variant appears, it can be tested based on the historical information collected from the cases where there is a history of failed test cases reused among variants. In this way, mitigating the problem of beginning without knowledge (learning) and adapting to changes in the execution environment, either by test case volatility or variant volatility.

6 CONCLUSION

This work investigates the use of *Multi-Armed Bandit* to address the *TCPCI* problem. Firstly, we performed a systematic mapping on this subject to obtain a general overview, and to identify challenges faced in this context, gaps, and trends to guide our work and the research community. Based on the findings observed in the mapping, we proposed a *MAB*-based approach, namely *COLEMAN* (*Combinatorial vOlatiLE Multi-Armed BANdiT*).

COLEMAN formulates the *TCPCI* problem as a *MAB* problem. Such a formulation considers that a test case is an arm to be pulled. However, due to the dynamic nature of our problem, two *MAB* variations are needed: combinatorial and volatile. At each turn (commit/build/*CI* cycle), the combinatorial variation allows a *MAB* policy to select all arms (test cases) available instead of one. The prioritization is defined based on the order in which the policy selects the arms. On the other hand, the volatile variation aims to figure out the test case volatility. For this, only the test set available at each commit is used by the policy.

The proposed approach is lightweight, that is, it needs only the historical failure data to prioritize a test case set. Moreover, no further detail about the system under test is needed, such as code coverage or code instrumentation. *COLEMAN* acts in the *CI* pipeline after a successful build and before test execution during the testing stage, prioritizing the test case set available. *COLEMAN* learns with feedback obtained from past prioritizations performed (online learning). Moreover, *COLEMAN* is a generic *MAB*-based approach, that is, we can use different *MAB* policies and reward functions.

To figure out the *COLEMAN* applicability in real scenarios, we performed five studies for assessing *COLEMAN* feasibility. In these studies, we considered a broad number of real-world software systems, quality indicators, different configuration settings, three time budgets, and different domains such as *HCS* context. *COLEMAN* was compared against different approaches such as *RETECS*, the state-of-the-art approach, search-based and deterministic approaches. The outcomes were extracted from the solutions analysis obtained in ≈ 42 thousand executions, disregarding parameter settings.

Comparing *COLEMAN* against *RETECS*, our approach presents better performance in the following indicators: *NAPFD* in $\approx 82\%$ of the cases, *NTR* in $\approx 73\%$, and *RFTC* in 100% ; independently of the reward functions and budgets investigated. On the other hand, both approaches present similar results considering *APFDc* metric. In relation to the execution time, *COLEMAN* is more stable, that is, adapts better to deal with peak of faults.

Furthermore, comparing *COLEMAN* solutions with the solutions provided by a *GA* approach, it provides near-optimal solutions in $\approx 90\%$ of the cases. Considering the execution time and the worst case, *COLEMAN* spends less than one second to prioritize a test suite; meanwhile, *GA* spends ≈ 31 seconds.

Finally, in comparison with a deterministic approach, *COLEMAN* obtained reasonable solutions in 92% of the cases and *RETECS* in 75% . Both approaches generated solutions that are very close to the optimal ones. Our findings allowed us to define guidelines for applying the approaches, identify approaches' limitations and improvements, and benchmark construction.

Regarding guidelines for application, *COLEMAN* is indicated in the great majority of cases, mainly for the systems that can be considered hard cases. On the other hand, *RETECS* is indicated in a restrictive budget scenario regarding early fault detection with cost consideration. Both approaches obtained more reasonable solutions using the reward function

RNFail. Considering accuracy, *COLEMAN* obtained best performance using *TimeRank*, and *RETECS* using *RNFail*.

In relation to the limitations and improvements, both approaches have some limitations to learn with few historical test data. Such a limitation is related to systems with a small number of *CI* cycles, peaks of failures, and a significant test case set, in which many failures are distributed over many test cases. Based on this drawback, a possible research direction is to design a hybrid approach that uses an algorithm with good performance with little historical data. Such an algorithm can be used to overcome the limitation in the first commits, and after enough information, *RETECS* or *COLEMAN* can be used.

Regarding benchmark construction, challenging prioritization cases are related to systems with high volatility regarding the number of test cases, peaks of failures, and the high number of failures distributed over many test cases. On the other hand, time reduction is more challenging in systems with low failure distribution over the test cases, and that present a small number of peaks in a few *CI* cycles, and failing test cases varying in each *CI* cycle.

Considering the evaluation conducted, we can accept the main hypothesis of this work presented in the introduction. The results endorse the use of a *MAB*-based approach to address the *TCPCI* problem, contributing efficiently and effectively.

6.1 LIMITATIONS

This section presents the limitations of the proposed approach. These limitations will be addressed in future work.

We evaluated *COLEMAN* with two reward functions to provide a fair comparison against *RETECS*. We have not explored other reward functions [87, 88, 98, 101] that could provide better results. Moreover, we evaluated our approach against a few approaches, but there are other approaches in the literature. A comprehensive evaluation is required to provide a better analysis of our approach.

COLEMAN is only applied in a controlled experiment scenario. The effectiveness of our approach is unclear, since we have not performed experiments in an industrial scenario.

Our approach addressed a few *CI* particularities. We have not explored other *CI* testing problems, such as Flaky Tests and UI testing. Besides that, the systems evaluated do not present the situation with multiple test requests (multiple commits). Another *CI* particularity not addressed is concerning the parallel test execution.

As the resources in *CI* environments are limited and an approach needs to spend little time to prioritize the test set, a limitation of our approach is related to the use of the Graphics Processing Unit. We did not evaluate this kind of strategy with *COLEMAN*. Such a strategy could speed up our approach regarding the prioritization process.

COLEMAN is based on historical information. A drawback regarding this kind of approach is the impact when we have a lack of data. Furthermore, *COLEMAN* uses only historical failure data to prioritize the most prominent tests, and other information related to tests can be used to prioritize them. For instance, test case duration and detection when a test case is change.

6.2 CONTRIBUTIONS

This work has the main following contributions:

- A systematic mapping about *Test Case Prioritization in Continuous Integration environments*;

- A model-free and lightweight *MAB*-based approach capable of automate the test case prioritization in *CI* environments, while learns with previous prioritization;
- An in-depth investigation of *Multi-Armed Bandit* in the *Test Case Prioritization in Continuous Integration environments* context considering real-world software systems;
- Comparison against a search-based algorithm in the *Test Case Prioritization in Continuous Integration environments* context;
- Learning-based strategies for Test Case Prioritization in Continuous Integration of Highly-Configurable Software;
- New quality indicators, benchmark, and guidelines for future research in the *Test Case Prioritization in Continuous Integration environments*;
- Supplementary Material for all studies conducted:
 - Jackson A Prado Lima and Silvia R Vergilio. Supplementary Material - Test Case Prioritization in Continuous Integration Environments: A Mapping Study. DOI: 10.17605/OSF.IO/ZFE64, 2020 [74].
 - Jackson A. Prado Lima and Silvia R. Vergilio. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration environments. DOI: 10.17605/OSF.IO/WMCBT, 2020 [70].
 - Jackson A. Prado Lima and Silvia R. Vergilio. Supplementary Material - Multi-Armed Bandit Test Case Prioritization in Continuous Integration Environments: A Trade-off Analysis. DOI: 10.17605/OSF.IO/J67EB, 2020 [73].
 - Jackson A. Prado Lima and Silvia R. Vergilio. Supplementary Material - An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration. https://osf.io/x96fk/?view_only=020b612cbdd84fa38d6a974743f9d823, 2021 [78].
 - Jackson A. Prado Lima, Willian D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. Supplementary material - learning-based prioritization of test cases in continuous integration of highly-configurable software. DOI: 10.17605/OSF.IO/5CD8M, 2020 [66].
 - Jackson A. Prado Lima, Willian D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. Supplementary material - cost-effective learning-based strategies for test case prioritization in continuous integration of highly-configurable software. https://osf.io/z3r2e/?view_only=db9ab0ed2b8e4289b22d4ad0c83c13c1, 2021 [67].

6.3 FUTURE WORK

Although we had good results, there is space for improvements. The next direction for future research includes the application of *COLEMAN* in other systems and possibly in an industrial scenario (case study). Moreover, the study of other policies to consider the context surrounding each *CI* cycle can give us relevant information to improve the prioritization. For instance, the test case duration can be used to execute the maximum of tests in a given time budget. Such kinds of policies are related to *Contextual Multi-Armed Bandit (CMAB)*.

We intend to evaluate the scalability of our approach and provide a relationship between the number of test cases and the prioritization time. Besides that, we intend to refine our approach concerning an ideal configuration by using irace [52] as the tuner. Furthermore, compare *COLEMAN* against to other techniques and reward functions available in the literature. Another work to be conducted is to provide *COLEMAN* as an API or an *add-on* to allow the use with the current *CI* tools available.

Regarding *HCS* context, we should conduct new evaluations applying the strategies proposed for other *HCS*s from different domains and with different number of variants. We also intend to apply the strategies using *COLEMAN* with other policies, as well as other features such as test coverage, testers' preference, test case duration and, to consider in how many variants a test case fails. Specific metrics to the variants could be explored.

A possible research direction is to investigate ways to adapt or select a *HCS* strategy according to some characteristics of the systems, for instance, related to the failure distribution. To this end, meta-learning techniques could be used.

Finally, we intend to provide *COLEMAN* as an-open source software to allow the research community to adopt, adapt, and improve the current study.

REFERENCES

- [1] Venkatachalam Anantharam, Pravin Varaiya, and Jean Walrand. Asymptotically Efficient Allocation Rules for the Multiarmed Bandit Problem with Multiple Plays-Part I: I.I.D. Rewards. *IEEE Transactions on Automatic Control*, 32(11):968–976, November 1987.
- [2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2):235–256, May 2002.
- [3] Maral Azizi. A Tag-based Recommender System for Regression Test Case Prioritization. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, ICSTW, pages 146–157, 2021.
- [4] Moritz Beller, Georgios Gousios, and Andy Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.
- [5] José Manuel Benitez, Juan Luis Castro, and Ignacio Requena. Are Artificial Neural Networks Black Boxes? *Transactions on Neural Networks*, 8(5):1156–1164, September 1997.
- [6] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *42nd International Conference on Software Engineering*, ICSE’20, New York, NY, USA, 2020. ACM.
- [7] Zahy Bnaya, Rami Puzis, Roni Stern, and Ariel Felner. Volatile Multi-Armed Bandits for Guaranteed Targeted Social Crawling. In *Proceeding of Workshops at the 27th AAAI Conference on Artificial Intelligence (Late-Breaking Developments)*, pages 8–10, 2013.
- [8] Nataniel P. Borges, Jr., Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft’18, pages 133–143. ACM, 2018.
- [9] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. Droidmate-2: A platform for android test generation. In *Proceedings of the 33rd International Conference on Automated Software Engineering*, ASE, pages 916–919. ACM, 2018.
- [10] Sébastien Bubeck and Nicolò et al. Cesa-Bianchi. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.
- [11] Buildbot. The Continuous Integration Framework. <https://buildbot.net>. Accessed: 2018-01-22.
- [12] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

- [13] Benjamin Busjaeger and Tao Xie. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 975–980, New York, NY, USA, 2016. ACM.
- [14] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, Sep 2013.
- [15] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. Build System with Lazy Retrieval for Java Projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 643–654, New York, NY, USA, 2016. ACM.
- [16] Younghwan Cho, Jeongho Kim, and Eunseok Lee. History-Based Test Case Prioritization for Failure Information. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference*, APSEC, pages 385–388, December 2016.
- [17] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [18] Christian Degott, Nataniel P. Borges Jr., and Andreas Zeller. Learning User Interface Element Interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, pages 296–306, New York, NY, USA, 2019. ACM.
- [19] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. An Empirical Study of the Effect of Time Constraints on the Cost-benefits of Regression Testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT ’08/FSE-16, pages 71–82, New York, NY, USA, 2008. ACM.
- [20] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
- [21] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, May 2001.
- [22] Sebastian Elbaum, Andrew McLaughlin, and John Penix. The Google Dataset of Testing Results, 2014.
- [23] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.
- [24] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft’s distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 11–20. ACM, 2016.

- [25] Thiago do Nascimento Ferreira, Josiel Neumann Kuk, Aurora Pozo, and Silvia Regina Vergilio. Product Selection Based on Upper Confidence Bound MOEA/D-DRA for Testing Software Product Lines. In *Congress on Evolutionary Computation*, CEC, pages 4135–4142. IEEE, July 2016.
- [26] Thiago do Nascimento Ferreira, Jackson Antonio do Prado Lima, Andrei Strickler, Josiel Neumann Kuk, Silvia Regina Vergilio, and Aurora Pozo. Hyper-Heuristic Based Product Selection for Software Product Line Testing. *Computational Intelligence Magazine*, 12(2):34–45, May 2017.
- [27] Álvaro Fialho. *Adaptive operator selection for optimization*. PhD thesis, Université Paris Sud-Paris XI, 2010.
- [28] Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES’17, page 114–123, New York, NY, USA, 2017. ACM.
- [29] M. Friedman. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, 11(1):86–92, 1940.
- [30] Alessio Gambi, Zabolotnyi Rostyslav, and Schahram Dustdar. Improving Cloud-based Continuous Integration Environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE ’15, pages 797–798, Piscataway, NJ, USA, 2015. IEEE Press.
- [31] GoCD. Open Source Continuous Delivery and Release Automation Server. <https://www.gocd.org>. Accessed: 2018-01-22.
- [32] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR’13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.
- [33] Giovanni Guizzo, Mosab Bazargani, Matheus Paixao, and John H. Drake. A Hyper-heuristic for Multi-objective Integration and Test Ordering in Google Guava. In *Proceedings of the International Symposium on Search Based Software Engineering*, SSBSE, pages 168–174. Springer, 2017.
- [34] Giovanni Guizzo, Gian Mauricio Fritsche, Silvia Regina Vergilio, and Aurora Trinidad Ramirez Pozo. A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, GECCO’15, pages 1343–1350, New York, NY, USA, 2015. ACM.
- [35] Giovanni Guizzo, Silvia R. Vergilio, Aurora T.R. Pozo, and Gian M. Fritsche. A multi-objective and evolutionary hyper-heuristic applied to the Integration and Test Order Problem. *Applied Soft Computing*, 56:331–344, 2017.
- [36] Giovanni Guizzo, Silvia Regina Vergilio, and Aurora Trinidad Ramirez Pozo. Evaluating a Multi-objective Hyper-Heuristic for the Integration and Test Order Problem. In *Brazilian Conference on Intelligent Systems (BRACIS)*, pages 1–6, Nov 2015.

- [37] Alireza Haghighatkah, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. Test prioritization in continuous integration environments. *Journal of Systems and Software*, 146:80–98, 2018.
- [38] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, Jan 2004.
- [39] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing White-box and Black-box Test Prioritization. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 523–534, New York, NY, USA, 2016. ACM.
- [40] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 426–437, New York, NY, USA, 2016. ACM.
- [41] Integrity. Continuous Integration Server. <https://integrity.github.io>. Accessed: 2018-01-22.
- [42] Jenkins. <https://wiki.jenkins-ci.org/display/JENKINS/Home>. Accessed: 2018-01-22.
- [43] Bo Jiang and Wing Kwong Chan. Testing and Debugging in Continuous Integration with Budget Quotas on Test Executions. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security, QRS*, pages 439–447. IEEE, August 2016.
- [44] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018.
- [45] Jung-Min Kim and Adam Porter. A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 119–129, New York, NY, USA, 2002. ACM.
- [46] William H. Kruskal and W. Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.
- [47] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.
- [48] Marco Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, Mar 2015.
- [49] Ke Li, Kalyanmoy Deb, Qingfu Zhang, and Sam Kwong. An Evolutionary Many-Objective Optimization Algorithm Based on Dominance and Decomposition. *Transactions on Evolutionary Computation*, 19(5):694–716, Oct 2015.

- [50] Ke Li, Álvaro Fialho, Sam Kwong, and Qingfu Zhang. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on*, 18(1):114–130, 2014.
- [51] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. Redefining Prioritization: Continuous Prioritization for Continuous Integration. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, pages 688–698, New York, NY, USA, 2018. ACM.
- [52] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. The irace package: Iterated Racing for Automatic Algorithm Configuration. *Operations Research Perspectives*, 3:43–58, 2016.
- [53] Mashael Maashi, Ender Özcan, and Graham Kendall. A multi-objective hyper-heuristic based on choice function. *Expert Systems with Applications*, 41(9):4475–4493, 2014.
- [54] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.
- [55] Dusica Marijan. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS '15*, pages 157–162. IEEE Computer Society, 2015.
- [56] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *2013 IEEE International Conference on Software Maintenance*, pages 540–543, September 2013.
- [57] Dusica Marijan and Marius Liaaen. Effect of Time Window on the Performance of Continuous Regression Testing. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 568–571. IEEE, October 2016.
- [58] Dusica Marijan, Marius Liaaen, Arnaud Gotlieb, Sagar Sen, and Carlos Ieva. TITAN: Test Suite Optimization for Highly Configurable Software. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 524–531, March 2017.
- [59] Dusica Marijan, Marius Liaaen, and Sagar Sen. DevOps Improvements for Reduced Cycle Times with Integrated Test Optimizations for Continuous Integration. In *Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference*, volume 01 of *COMPSAC*, pages 22–27. IEEE, July 2018.
- [60] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP '17*, pages 233–242, Piscataway, NJ, USA, 2017. IEEE Press.
- [61] David Moher, Alessandro Liberati, Jennifer Tetzlaff, and Douglas G Altman. Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement. *BMJ*, 339, 2009.

- [62] Armin Najafi, Weiyi Shang, and Peter C. Rigby. Improving test effectiveness using test executions history: An industrial experience report. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP, pages 213–222, Piscataway, NJ, USA, 2019. IEEE Press.
- [63] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE’08, pages 68–77, Swindon, UK, 2008. BCS Learning & Development Ltd.
- [64] Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–11, 2020.
- [65] Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. Cost-effective learning-based strategies for test case prioritization in Continuous Integration of Highly-Configurable Software. *Empirical Software Engineering*, 2021. Accepted.
- [66] Jackson A. Prado Lima, Willian D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. Supplementary material - learning-based prioritization of test cases in continuous integration of highly-configurable software. DOI: 10.17605/OSF.IO/5CD8M, 2020.
- [67] Jackson A. Prado Lima, Willian D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. Supplementary material - cost-effective learning-based strategies for test case prioritization in continuous integration of highly-configurable software. https://osf.io/z3r2e/?view_only=db9ab0ed2b8e4289b22d4ad0c83c13c1, 2021.
- [68] Jackson A. Prado Lima and Silvia R. Vergilio. A Multi-Objective Optimization Approach for Selection of Second Order Mutant Generation Strategies. In *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*, SAST. ACM, 2017.
- [69] Jackson A. Prado Lima and Silvia R. Vergilio. A systematic mapping study on higher order mutation testing. *Journal of Systems and Software*, 154:92–109, 2019.
- [70] Jackson A. Prado Lima and Silvia R. Vergilio. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration environments. DOI: 10.17605/OSF.IO/WMCBT, 2020.
- [71] Jackson A. Prado Lima and Silvia R. Vergilio. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, page 12, 2020.
- [72] Jackson A. Prado Lima and Silvia R. Vergilio. Multi-armed bandit test case prioritization in continuous integration environments: A trade-off analysis. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, SAST’20. ACM, 2020.
- [73] Jackson A. Prado Lima and Silvia R. Vergilio. Supplementary Material - Multi-Armed Bandit Test Case Prioritization in Continuous Integration Environments: A Trade-off Analysis. DOI: 10.17605/OSF.IO/J67EB, 2020.

- [74] Jackson A Prado Lima and Silvia R Vergilio. Supplementary Material - Test Case Prioritization in Continuous Integration Environments: A Mapping Study. DOI: 10.17605/OSF.IO/ZFE64, 2020.
- [75] Jackson A. Prado Lima and Silvia R. Vergilio. Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, 121:106268, 2020.
- [76] Jackson A. Prado Lima and Silvia R. Vergilio. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration environments. *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Aug 2021. Journal First Track. Presentation.
- [77] Jackson A. Prado Lima and Silvia R. Vergilio. An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration. *Journal of Software Engineering Research and Development*, 2021. Submitted.
- [78] Jackson A. Prado Lima and Silvia R. Vergilio. Supplementary Material - An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration. https://osf.io/x96fk/?view_only=020b612cbdd84fa38d6a974743f9d823, 2021.
- [79] Jackson A. Prado Lima and Silvia Regina Vergilio. Comparing low level heuristics selection methods in a higher-order mutation testing approach. In *Proceedings of the IX Workshop on Search Based Software Engineering*, WESB, 2018.
- [80] Jackson A. Prado Lima and Silvia Regina Vergilio. Search-Based Higher Order Mutation Testing: A Mapping Study. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*, SAST'18, page 87–96, New York, NY, USA, 2018. Association for Computing Machinery.
- [81] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *IEEE International Conference on Software Maintenance*, pages 255–264, October 2007.
- [82] Herbert Robbins. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*, pages 169–177. Springer, 1985.
- [83] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test Case Prioritization: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–. IEEE Computer Society, 1999.
- [84] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing Test Cases For Regression Testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, October 2001.
- [85] Enrique A. Roza, Jackson A. Prado Lima, Silvia R. Vergilio, and Rogério C. Silva. Machine Learning Regression Techniques for Test Case Prioritization in Continuous Integration Environment. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021. Approved.
- [86] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous Deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th*

- International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30, May 2016.
- [87] Ying Shang, Qianyu Li, Yang Yang, and Zheng Li. Occurrence Frequency and All Historical Failure Information Based Method for TCP in CI. In *Proceedings of the International Conference on Software and System Processes, ICSSP '20*, page 105–114, New York, NY, USA, 2020. Association for Computing Machinery.
 - [88] Tingting Shi, Lei Xiao, and Keshou Wu. Reinforcement Learning Based Test Case Prioritization for Enhancing the Security of Software. In *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 663–672, 2020.
 - [89] Henrique N. Silva, Jackson A. Prado Lima, Silvia R. Vergilio, and Andre T. Endo. A Mapping Study on Mutation Testing for Mobile Applications. *Software Testing, Verification and Reliability*, page 23, 2021.
 - [90] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017*, pages 12–22, New York, NY, USA, 2017. ACM.
 - [91] Andrei Strickler, Jackson A. Prado Lima, Silvia R. Vergilio, and Aurora T.R. Pozo. Deriving products for variability test of feature models with a hyper-heuristic approach. *Applied Soft Computing*, 49:1232–1242, 2016.
 - [92] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *The MIT Press*, 2011.
 - [93] Travis CI. Travis CI. <https://travis-ci.org>. Accessed: 2018-01-22.
 - [94] Andras Vargha and Harold D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, January 2000.
 - [95] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.
 - [96] Wen Wen, Zhongju Yuan, and Yuyu Yuan. Improving RETECS method using FP-Growth in continuous integration. In *Proceedings of the 5th IEEE International Conference on Cloud Computing and Intelligence Systems, CCIS*, pages 636–639. IEEE, Nov 2018.
 - [97] Woo, Maverick and Cha, Sang Kil and Gottlieb, Samantha and Brumley, David. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the SIGSAC Conference on Computer & Communications Security, CCS'13*, pages 511–522, New York, NY, USA, 2013. ACM.
 - [98] Zhaolin Wu, Yang Yang, Zheng Li, and Ruilian Zhao. A Time Window Based Reinforcement Learning Reward for Test Case Prioritization in Continuous Integration. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware, Internetware '19*, New York, NY, USA, 2019. Association for Computing Machinery.
 - [99] Lei Xiao, Huaikou Miao, Tingting Shi, and Yu Hong. LSTM-based deep learning for spatial-temporal software testing. *Distributed and Parallel Databases*, 38:687–712, May 2020.

- [100] Lei Xiao, Huaikou Miao, and Ying Zhong. Test case prioritization and selection technique in continuous integration development environments: a case study. *International Journal of Engineering & Technology*, 7(2.28):332–336, 2018.
- [101] Yang Yang, Zheng Li, Liulu He, and Ruilian Zhao. A systematic study of reward for reinforcement learning based continuous integration testing. *Journal of Systems and Software*, 170:110787, 2020.
- [102] Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.
- [103] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71, Oct 2017.
- [104] Li Zhou. A Survey on Contextual Multi-armed Bandits. *CoRR*, abs/1508.03326, 2015.
- [105] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. Test Re-Prioritization in Continuous Testing Environments. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME, pages 69–79. IEEE, September 2018.
- [106] Eckart Zitzler and Simon Künzli. Indicator-based selection in multiobjective search. In *Proceedings of the Parallel Problem Solving from Nature - PPSN VIII*, pages 832–842, Berlin, Heidelberg, 2004. Springer.
- [107] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: improving the strength pareto evolutionary algorithm. Technical report, Department of Electrical Engineering, Swiss Federal Institute of Technology, Zurich, Switzerland, 2001.
- [108] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. *CoRR*, abs/1611.01578, 2017.

APPENDIX A – TCPCI: A SYSTEMATIC MAPPING STUDY

Test Case Prioritization in Continuous Integration Environments: A Systematic Mapping Study

Jackson A. Prado Lima
Department of Informatics
Federal University of Paraná (UFPR)
Curitiba, Paraná, Brazil
japlima@inf.ufpr.br

Silvia R. Vergilio
Department of Informatics
Federal University of Paraná (UFPR)
Curitiba, Paraná, Brazil
silvia@inf.ufpr.br

ABSTRACT

Context: Continuous Integration (CI) environments allow frequent integration of software changes, making software evolution more rapid and cost-effective. In such environments, the regression test plays an important role, as well as the use of Test Case Prioritization (TCP) techniques. Such techniques attempt to identify the test case order that maximizes certain goals, such as early fault detection. This research subject has been raising interest because some new challenges are faced in the CI context, as TCP techniques need to consider time constraints of the CI environments. *Objective:* This work presents the results of a systematic mapping study on Test Case Prioritization in Continuous Integration environments (TCPIC) that reports the main characteristics of TCPIC approaches and their evaluation aspects. *Method:* The mapping was conducted following a plan that includes the definition of research questions, selection criteria and search string, and the selection of search engines. The search returned 35 primary studies classified based on the goal and kind of used TCP technique, addressed CI particularities and testing problems, and adopted evaluation measures. *Results:* The results show a growing interest in this research subject. Most studies have been published in the last four years. 80% of the approaches are history-based, that is, are based on the failure and test execution history. The great majority of studies report evaluation results by comparing prioritization techniques. The preferred measures are Time and number/percentage of Faults Detected. Few studies address CI testing problems and characteristics, such as parallel execution and test case volatility. *Conclusions:* We observed a growing number of studies in the field. Future work should explore other information sources such as models and requirements, as well as CI particularities and testing problems, such as test case volatility, time constraint, and flaky tests, to solve existing challenges and offer cost-effective approaches to the software industry.

KEYWORDS

Software Testing, Continuous Integration, Test Case Prioritization

1 INTRODUCTION

With the adoption of the agile paradigm by most software organizations, we observe a growing interest in Continuous Integration (CI) environments. Such environments allow more frequent integration of software changes, making software evolution more rapid and cost-effective [40]. This because they automatically support tasks like build process, test execution, and test results report. The results are used to resolve problems and locate faults, and rapid feedback is fundamental to reduce development costs [PS11].

Within an integration cycle (usually called build), regression testing is an activity that takes a significant amount of time. A test set, many times, includes thousands of test cases whose execution takes several hours or days [PS9].

To help in the regression testing task, we find in the literature some techniques, which are usually classified into three main categories [38]: minimization, selection, and prioritization. Techniques based on Test Case Minimization (TCM) usually remove redundant test cases, minimizing the test set according to some criterion. Test Case Selection (TCS) selects a subset of test cases, the most important ones to test the software. Test Case Prioritization (TCP) attempts to re-order a test suite to identify an “ideal” order of test cases that maximizes specific goals, such as early fault detection. TCP techniques are very popular in the industry and are the focus of our work.

Existing TCP techniques can be applied in CI environments. However, most of them require adaptations to deal with the testing budget, limited resources, and constraints of the CI environments. For instance, the application of search-based techniques or other ones that require extensive code analysis and coverage may be unfeasible due to the time constraints and the test budget for a build.

We can see that the intersection of TCP and CI (TCPIC - *Test Case Prioritization in Continuous Integration environments*) poses some difficulties. Such a subject has been raising interest, and approaches to solve the problem have been proposed in the literature. However, we have not found works reporting the characteristics of such approaches, if they differ from existing TCP approaches, and how they have been evaluated. To provide such a general overview

is important to identify challenges faced, gaps, and trends to guide research on this subject and to propose new ways to perform regression testing within integration cycles. Motivated by these facts, this work presents the results of a mapping study on TCPCI.

We adopted the process of Petersen et al. [27]. We followed a research plan including research questions, inclusion and exclusion criteria, construction of the search string, and selection of known search databases. We found 35 primary studies, and we analyzed the results according to three aspects: i) *basic information of the field*, such as evolution along the years, main authors and publication fora; ii) *characteristics of the approaches*: main goal, kind of used TCP technique, programming language, addressed CI testing problems such as flaky and time-consuming tests, as well as some CI particularities related to the testing budget and constraints, parallelism, multiple test requests, and volatility of test cases; and iii) *evaluation aspects*: such as evaluation contexts and measures used for comparison. In addition to this, some research trends and gaps were identified that allow researchers to direct future investigations.

The results show an increasing number of papers on this research subject. Most studies have been published in the last four years. 80% of the approaches are history-based, that is, are based on the failure and test execution history. The great majority of studies report evaluation results by comparing prioritization techniques, and we also found studies that use TCP after (or in combination) with TCS. The preferred measures are Time and number or percentage of Faults Detected (FD). Few studies consider CI characteristics and testing problems.

Then the contribution of this mapping is twofold: i) to present the main characteristics of the TCPCI approaches according to a classification schema generated interactively during the analysis of the studies found; and ii) to help researchers in the identification of research opportunities and encourage new works on this subject to solve existing challenges and offer to the software industry cost-effective approaches.

The remaining of this paper is organized as follows. Section 2 overviews background and related work. Section 3 describes the adopted mapping process. Section 4 analyses the results. Section 5 points out trends and identified research opportunities. Section 6 discusses threats to validity. Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

This section first introduces the fields explored in our mapping: Test Case Prioritization (TCP) and Continuous Integration environments, and after, reviews related work.

2.1 Test Case Prioritization

According to Rothermel et al. [31] the TCP problem can be formulated as:

Definition 2.1. Given T , a test suite, PT , the set of all possible permutations of T , and f , a function that determines the performance of a given prioritization from PT to real numbers. Find:

$$T' \in PT \text{ s.t. } (\forall T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$$

TCP problem aims to find the best T' that achieves certain specific goals [31].

This technique allows the most crucial test cases are first executed. This characteristic is interesting (and suggested), given that in a regression testing scenario, there are limited resources, and it may not be possible to execute the entire regression test suite [31, 38]. TCP techniques consider all the test suite and can be, in some cases, time-consuming. But such a characteristic is also an advantage because the coverage is maintained. Besides, in the presence of time and cost constraints for the test activity that hinders the execution of all test cases, TCP allows failing test cases are executed first. This contributes to reduce resources and costs spent in the test, as well as ensures that the maximum possible fault coverage is achieved.

According to Rothermel et al. [31], some objectives of TCP techniques are: (i) to increase the fault detection rate of a test suite already in the beginning of the regression test execution; (ii) to increase the system code coverage under test; (iii) to increase high-risk fault detection rate; and (iv) to increase the probability to reveal faults related with specific code changes. To achieve such objectives, many TCP techniques were proposed in the literature. According to the information used in the prioritization, they can be classified into different categories [38] that we use to categorize the approaches found in our mapping. They are: Cost-aware, Coverage-based, Distribution-based, Human-based, History-based, Requirement-based, Model-based, and Probabilistic. An overview of these categories is presented in Section 3.5, which describes our classification schema.

2.2 Continuous Integration Environments

In the past, some developers worked alone (separately) for a long time and only integrated their changes to the master branch when they completed their work. However, this practice presents some disadvantages. It is time-consuming, adds unnecessary bureaucratic cost to the projects, and results in the accumulation of uncorrected errors for long periods. These factors hampered a rapid distribution of updates to customers.

With the advent of the agile development paradigm, Continuous Software Engineering practices have become popular and adopted by most organizations, such as Continuous Integration (CI), Continuous Deployment (CD), and Continuous DELivery (CDE), allowing frequent integration, test of the changed code, deployment, and quick feedback from the customer in a very rapid cycle. The relationship between such practices is illustrated in Figure 1.

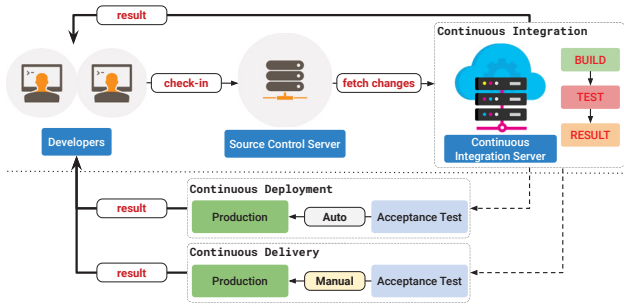


Figure 1: Overview of the relationship between Continuous Integration, Delivery, and Deployment (adapted from Shahin et al. [32]).

CI is an essential practice adopted before deployment and delivery. CI environments contain automated software building and testing [23], helping to scale up headcount and delivery output of engineering teams, as well as allowing software developers to work independently on features in parallel. When they are ready to merge these features into the end product, they can do this independently and rapidly. Among the popular open-source CI servers we can mention: Buildbot [3], GoCD [13], Integrity [15], Jenkins [17], and Travis CI [35].

CDE aims at packing an artifact (the production-ready state from the application) to be delivered for Acceptance Test. In this sense, the artifact should be ready to be released to end-users (production) at any given time. On the other hand, CD is responsible for automatically packing, launching and distributing the software artifact to production. We can observe that CI ensures that the artifact used in the CD and CDE successfully passed the integration phase.

In order to provide a swift and cost-effective way to validate and launch new software updates, improving fault detection and software quality, regression testing is an essential activity in CI environments. This is because to enable rapid test feedback in CI, test cycles are restricted to a specific (short) duration. We refer to this time duration of each test cycle as a time budget. Time budgets can vary from a cycle to a cycle, and they include time to select relevant tests for running, to run the tests, and to report test results to developers. In this way, traditional TCP techniques require some adaptations to be applied in the CI context. The techniques need to consider specific particularities of CI environments like as: parallel execution of test cases and allocation of resources, the volatility of test cases, that is, the test cases can be added and removed in subsequent commits, the detection of as many faults as possible in a short interval of time. To address all these particularities, some approaches have emerged in the literature, being TCPCI an emergent research topic that motivates our mapping. 3

2.3 Related Work

In the literature we can find some mappings, surveys and systematic reviews on continuous integration practices and environments [19, 32], on regression testing [38], test case prioritization techniques in general [4, 20, 21, 33], and on software testing in agile and continuous integration context [5, 6, 25, 34]. Such works do not specifically address the subject test case prioritization in continuous integration environments. The work of Shahin et al. [32] mentions some studies concerning TCP that were included in our mapping, however, the focus of their work is neither TCP techniques nor regression testing techniques.

We summarize the key aspects of the abovementioned studies in Table 1, which presents, for each work, the corresponding research focus, type of study, the number of primary studies (NS) found, the publication year, and the period covered. In short, our work differs from existing ones because our focus is on TCPCI. Our mapping offers details and characteristics of the TCPCI studies, presents research gaps and trends that can help the researchers in future investigations. We can not find such results in the works most related to ours, which map the general areas in a separated way.

3 MAPPING PROCESS

In this section, we describe the main steps of our mapping, conducted following the guidelines proposed by Petersen et al. [27].

First of all, and to justify our mapping, we conducted a search for related work with the following string: *test AND prioritization AND (map OR mapping OR review OR survey)*. We did not find work with the same goal of ours. Given this fact, which justifies and shows the need and relevance of our mapping, we performed the steps described next.

3.1 Definition of Research Questions

To overview existing research on TCPCI, we used three groups of research questions. The first one provides basic information about the primary studies. The second group characterizes the approaches found in the studies. The third group refers to the evaluation aspects of each proposal. The research questions from each group are presented below.

RQ1: Basic information of the field

RQ1.1: *In which fora is the research on TCPCI published?*

This question allows the identification of the target public, as well as the main fora where research on TCPCI can be found and published.

RQ1.2: *How have the number and frequency of publications evolved over the years?* This question aims to analyze the interest in TCPCI over the years and to assess how relevant and active this topic

Table 1: Comparison of our work with existing secondary studies.

Work	Focus	Type	NS	Pub. Year	Period Covered
Hellmann et al. [5]	Agile Testing	Mapping	166	2012	2003-2011
Eck et al. [6]	CI	Review	43	2014	Not Identified
Ståhl and Bosch [34]	CI	Review	46	2014	2003-2013
Mäntylä et al. [25]	Rapid Release	Review	24	2015	2003-2013
Karvonen et al. [19]	Agile Release	Review	71	2017	2005-2015
Shahin et al. [32]	CI	Review	69	2017	2004-2016
Yoo and Harman [38]	TCM, TCS, and TCP	Survey	159	2012	1977-2009
Singh et al. [33]	TCP	Review	65	2012	1997-2011
Catal et al. [4]	TCP	Mapping	120	2013	2001-2011
Kumar and Singh [21]	TCP	Survey	11	2014	2001-2013
Khatibsyarbini et al. [20]	TCP	Review	80	2018	1999-2016
Our work	TCP and CI	Mapping	35	2019	2009-2019

is, as well as the maturity of the research being conducted.

RQ1.3: *What are the main research groups?* This question aims to identify the main research groups as well as authors and countries.

RQ2: Characteristics of the approach

RQ2.1: *What types of approaches are proposed?* As mentioned in Section 2, the approaches can be classified according to the information used in the prioritization. This question aims to identify the most common type of TCP/CI approach and research opportunities.

RQ2.2: *What are the application contexts?* This question aims to identify application aspects related to: the CI environment particularities, addressed testing problems, programming language, assumptions, requirements, and limitations. This question aims to identify possible research gaps and to point the need for future work.

RQ3: Evaluation Aspects

RQ3.1: *How have the approaches been evaluated?* This question analyses different aspects of the conducted evaluation, characterizing: used systems and datasets, threats to validity, baseline approach used in the comparison, etc. The evaluation measure is an important aspect that is treated in a separate question.

RQ3.2: *What are the used evaluation measures?* This question identifies all the measures used in the evaluation and whether their values are statistically analyzed. Most measures are based on the fault detection ability, but in the context of CI environments, the runtime is an important aspect to be evaluated.

3.2 Definition of the Search String

We formulated our search string, considering the mapping goals, and posed research questions. The resulting search terms were composed of synonymous for the main terms “Test Prioritization” and “Continuous Integration” taking into account different spelling. Then, we constructed the search string using logical operators, “OR” and “AND”. After tests, we adopted the following search string that returned the greatest number of relevant articles.

(“continuous integration”) **AND** (prioritization OR prioritisatio) **AND** (test OR testing)

To verify the accuracy of the keywords chosen to compose the search string, we used a control group. Such a group comprises a set of previously known studies, which are presented in Table 2 along with the number of citations extracted from Google Scholar on 1st October 2019.

Table 2: Control group.

Year	Authors	Title	Citations
2009	Jiang et al. [PS14]	How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization	79
2012	Jiang et al. [PS13]	How Well Does Test Case Prioritization Integrate with Statistical Fault Localization?	41
2013	Marijan et al. [PS21]	Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study	73
2014	Elbaum et al. [PS7]	Techniques for Improving Regression Testing in Continuous Integration Development Environments	157
2016	Busjaeger and Xie [PS4]	Learning for Test Prioritization: An Industrial Case Study	27
2017	Spieker et al. [PS28]	Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration	33
2018	Liang et al. [PS18]	Redefining Prioritization: Continuous Prioritization for Continuous Integration	11
2018	Haghighatkah et al. [PS9]	Test prioritization in continuous integration environments	4

3.3 Selection criteria

Table 3 presents the adopted inclusion and exclusion criteria. In summary, we screened studies written in English, available online, and verifying each one that fits the goals of our mapping.

Table 3: Inclusion and exclusion criteria.

Inclusion Criteria	
I1	The paper is related to Software Engineering area;
I2	The paper is related to TCPCI.
Exclusion Criteria	
E1	Out of scope, not related to TCPCI;
E2	Not available online;
E3	Not in English;
E4	Abstracts, posters, reviews, conference reviews, chapters, thesis, keynotes, and doctoral symposiums.

3.4 Conducting the review

The search started and finished in October 2019. The review was conducted in a set of steps, presented in Figure 2 following the PRISMA (Preferred Reporting Items for Systematic reviews and Meta-Analyses) statement [26]. In such a figure, we present three data sources used in our mapping: Digital Libraries, Grey Literature, and Snowballing.

First, we selected primary studies by using the search string in the digital libraries, considering the title, abstract, and keywords. The engines for performing the search are online repositories that were chosen due to their popularity and because they provide many leading software engineering publications. However, some of them required adaptations, for example, searching in title, abstract, and keywords individually.

In order to identify the state-of-the-art and -practice in the subject of this mapping, we included a search in the grey literature [11, 12] (non-published, nor peer-reviewed sources of information) seeking relevant white papers, industrial (and technical) reports. However, the use of grey literature can lead to unreasonably grow in the number of studies. In addition to this, the quality assessment of the studies found in the grey literature is more difficult. Because of this, we reduced the possible noise in the search, i.e. ignoring theses and incomplete studies, once that there is a high possibility that a thesis produces articles as results. Additionally, we performed a snowballing reading (forward and backward snowballing procedures, following Wohlin's guidelines [36]) by searching in the control group. In this step, we found five papers.

We did not define an initial publication date for the studies, then all the returned papers were included. Table 4 presents the data sources used and the period covered,

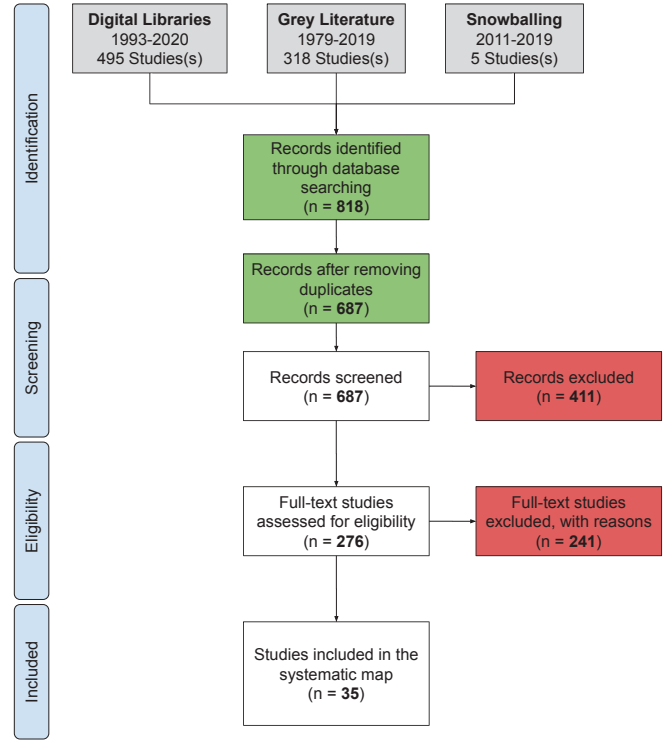


Figure 2: PRISMA flow diagram. For each data source, the number of studies found is presented, as well as the range of publication data from these studies.

with the data separated by Digital Libraries, Grey Literature, and Snowballing. In total, we performed the mapping in 17 data sources. As we can see, a total of 818 studies were found, including the period from 1979 to 2020.

In the second step, we removed repeated studies, remaining 687 ones. Then we applied the selection criteria and obtained 276 studies. The selection was conducted by the first author, who read the title, abstract, and index terms of the papers (keywords). In case of doubts, reading in the following order was performed: introduction, conclusion, and the entire paper. If such a doubt persists, meetings and discussions with the second author were accomplished to solve it. The second author also revised the decisions of the first one. In the end, 35 papers remain. The selected papers are presented in the end of this paper (*Primary Studies (PS) Section*).

3.5 Classification scheme, data extraction and dissemination

We collected the following information to address the research questions: title, authors, institution, publication venue, publication year, and depending on the study type (theoretical or experimental) other data were collected such

Table 4: Number of papers returned by each source.

Source	Studies	Period Covered
Digital Libraries		
ACM	24	2009-2019
EBSCOhost	3	2012-2018
Ei Compendex	31	2009-2019
IEEEExplore	20	2009-2019
ISI Web of Science	27	2009-2019
MIT Libraries	20	2009-2019
ScienceDirect	2	2012-2018
Scopus	35	2009-2019
Springer Link	273	1993-2020
Wiley Online Library	60	2007-2019
Grey Literature		
AWS Whitepapers & Guides	0	-
Google AI	0	-
Google Cloud	2	2018-2019
Google Scholar	129	2001-2019
Microsoft Academic	2	2009-2018
Microsoft Research	0	-
Science.gov	185	1979-2019
Snowballing		
Forward and Backward procedures	5	2011-2019
Total	818	1979-2020

as all the characteristics of the techniques used by the approach and conducted evaluation. The raw data were analyzed and disseminated in the Open Science Framework (OSF) [28].

In order to avoid the researchers' bias threat during the extraction, which can affect negatively the results, we used a classification scheme, derived interactively, which encompasses four dimensions to classify the studies found. The schema is detailed in Table 5 with a brief description of each category. Regarding the characteristics of the approach (RQ2), we use the dimensions: research goal, information source, and CI testing problems. The categories used in the first dimension were adapted from the classification proposed by Catal and Mishra [4]. In the second dimension, the categories were extracted from the work of Yoo and Harman [38], and in the third one, we used the problems reported by Laukkanen et al. [22], which are more related to test in CI environments. To answer RQ3, we considered each measure used in the evaluation as a category.

4 OUTCOMES

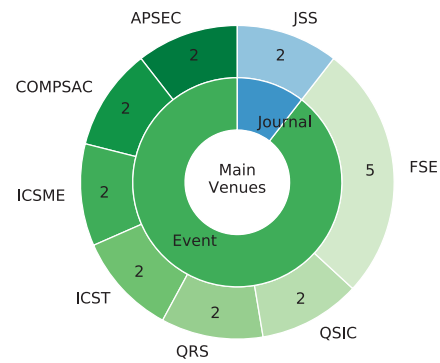
This section presents the results of our mapping and analysis to answer our research questions. To this end, we derived Table 6 containing for each study a summary of our main findings: reference and year of the study, research

goal, information considered by the approach, and used evaluation measure.

4.1 Basic Information of the field

This section presents answers to the questions concerning basic Information of the field: Section 4.1.1 answers RQ1.1, Section 4.1.2 answers RQ1.2, and Section 4.1.3 answers RQ1.3.

4.1.1 RQ1.1 - Publication Fora. We analyzed the publication fora, and we observed that 28 papers (80%) were published in events, and 7 papers in journals. We found 24 different publication venues. Most of them (16 ($\approx 67\%$)) published only one paper, and 7 (29%) published two papers. The preferred journal is the Journal of Systems and Software (JSS) with two papers. Among the events that published more than one paper, we can mention: Asia-Pacific Software Engineering Conference (APSEC), International Computer Software and Applications Conference (COMPSAC), International Symposium on Foundations of Software Engineering (FSE), International Conference on Software Maintenance and Evolution (ICSME), International Conference on Software Testing, Verification and Validation (ICST), International Conference on Software Quality, Reliability and Security (QRS), and International Conference on Quality Software (QSIC). Figure 3 shows the distribution of main venues: journals and events with more than two papers. The preferred fora is the FSE with five papers. We can see that the venues are all related to the general area of software engineering and, specifically, with software quality and maintenance.

**Figure 3: Main publication fora with more than two papers.**

4.1.2 RQ1.2 - Publications over the years. The number of publications over the years is depicted in Figure 4 together with a trend-line. This figure also contains bars representing the venue (Journal or Event). The trend line shows a crescent interest in the topic. The first paper appeared in

Table 5: Classification schema.

Dimension/Category	Description
RQ2	
Research Goal	
Prioritization	The study proposes a prioritization technique.
Measurement	The study proposes an evaluation measure to evaluate prioritization technique.
Comparison	The study compares prioritization techniques.
Minimization	The study combines prioritization and minimization techniques.
Selection	The study combines prioritization and selection techniques.
Localization	The study uses the prioritization technique for fault localization.
Time-Limit Treatment	The study considers time constraint in Continuous Integration environments.
Others	The study has a different goal not included in the other categories.
Information Source	
Cost-aware	The approach prioritizes test cases based on the cost of the test cases, because their costs cannot be equal.
Coverage-based	The approach prioritizes test cases based on the code coverage.
Distribution-based	The approach prioritizes test cases based on the distribution of the test case profiles.
Human-based	The approach prioritizes test cases based on factors that (human) testers deem the most important.
History-based	The approach prioritizes test cases based on test case execution history information and code changes.
Requirement-based	The approach prioritizes test cases based on information extracted from requirements.
Model-based	The approach prioritizes test cases based on information extracted from models, such as UML (Unified Modeling Language) models.
Probabilistic	The approach prioritizes test cases based on probabilistic theories.
Others	The approach prioritizes test cases based on other kind of information not included in the other categories.
CI Testing Problem	
Flaky tests	Tests that randomly fail sometimes.
Time-Consuming testing	Testing takes too much time.
User Interface (UI) testing	UI testing of the application.
Complex testing	Testing is complex, e.g., setting up the environment is complex.
RQ3	
Evaluation Measure	
APFD	Average Percentage of Faults Detected
APFDc	Average Percentage of Faults Detected per Cost
NAPFD	Normalized Average Percentage of Faults Detected
Expense	Minimum percentage of statements in a program that must be examined to locate the fault.
FD	Number or percentage of Faults Detected.
Time	Time spent to execute the prioritization method or the test suite.
NA	The study does not use an evaluation measure.
Others	The study uses other types of evaluation measure not mentioned above.

2009, and since then, the interest in TCPCI has been increasing. The most significant number of publications were found in 2018, with 9 studies out of 35 ($\approx 26\%$). Considering the search ended in October 2019, and that some studies of this year may not be included, we can see that 21 studies ($\approx 60\%$) were published in the period 2016-2018. This corroborates the fact TCPCI is an emergent research-topic that has been arising interest in the last years.

4.1.3 RQ1.3 - Research Groups. We found 83 authors from 45 different institutions. We observe that the main research groups are in the Simula Research Laboratory that appears in 8 (23%) different studies, followed by Cisco appearing in 6 (17%) studies. The University of Hong Kong and City University of Hong Kong appearing in 4 (11%) studies, Beihang University appearing in 3 (9%) studies, and Concordia University, Sungkyunkwan University, Peking University, Mälardalen University, University of Nebraska, Westermo Research and Development, and Google in 2 studies.

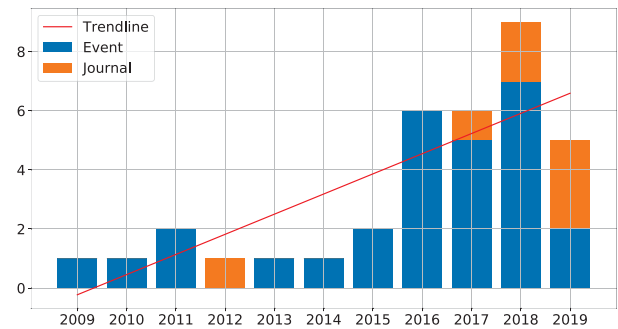


Figure 4: Number of publications over the years.

Table 7 shows the main authors with more than two studies published and the number of corresponding studies (NS). Whilst, we identified 12 different countries. Table 8 presents the countries found and the corresponding number of studies (NS).

Table 6: Main findings

This table presents the main findings for each study. The first column contains the reference of the study, followed by the year of publication in the second column. Columns 3 to 9 show the Research Goals of each study, where “P” means Prioritization, “C” Comparison, “S” Selection, “M” Minimization, “FL” Fault Localization, “TL” Time-Limit Treatment, and “O” Others. Columns 10 to 16 show the Considered Information during prioritization, where “Cost” means Cost-aware, “Cov” Coverage-based, “Dis” Distribution-based, “Hist” History-based, “Human” Human-based, “Prob” Probabilistic, and “O” Others. The last columns show the used Evaluation Measures (see Table 5), where “E” means Expense and “O” Others.

Ref.	Year	Research Goal					Considered Information					Evaluation Measure					O					
		P	C	S	M	FL	TL	O	Cost	Cov	Dis	Hist	Human	Prob	O	APFD		APFDc	NAPFD	E	FD	Time
[PS14]	2009	-	✓	-	-	✓	-	-	-	-	✓	✓	-	-	✓	✓	-	-	✓	-	-	✓
[PS10]	2010	-	✓	-	-	✓	-	-	-	-	✓	-	-	-	✓	-	-	-	✓	-	-	✓
[PS12]	2011	-	✓	-	-	✓	-	-	-	-	✓	-	-	-	✓	-	-	-	✓	-	-	✓
[PS33]	2011	✓	-	✓	✓	-	✓	-	-	-	✓	-	-	-	✓	-	-	-	-	-	✓	✓
[PS13]	2012	-	✓	-	-	✓	-	-	-	-	-	✓	-	-	✓	-	-	-	✓	-	-	✓
[PS21]	2013	✓	✓	-	-	-	✓	-	✓	-	-	-	✓	-	-	-	✓	-	-	✓	✓	-
[PS7]	2014	✓	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	-
[PS19]	2015	✓	✓	-	-	-	✓	-	✓	✓	-	✓	-	-	-	-	✓	-	-	-	-	-
[PS16]	2015	-	✓	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓
[PS22]	2016	-	✓	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	✓	-	-
[PS4]	2016	✓	✓	-	-	-	-	-	-	-	-	✓	-	-	✓	✓	-	-	-	-	-	-
[PS6]	2016	✓	✓	-	-	-	-	-	-	-	-	-	-	-	✓	✓	-	-	-	-	-	-
[PS8]	2016	-	-	✓	-	-	✓	-	-	-	-	-	-	-	✓	-	-	-	-	-	✓	✓
[PS30]	2016	✓	-	-	-	-	✓	-	-	-	-	-	✓	-	✓	-	-	-	-	✓	✓	✓
[PS11]	2016	-	✓	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-	-	✓	-	-	-
[PS28]	2017	✓	✓	-	-	-	✓	-	-	-	-	-	-	-	✓	-	-	✓	-	-	-	-
[PS24]	2017	-	✓	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-	-	✓	-	✓
[PS15]	2017	✓	✓	-	-	-	-	-	-	-	-	-	-	-	✓	✓	-	-	-	-	-	-
[PS23]	2017	✓	✓	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-	-	✓	✓	✓
[PS29]	2017	✓	-	-	-	-	✓	-	-	-	-	-	✓	-	✓	-	-	-	-	✓	✓	✓
[PS17]	2017	✓	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	✓
[PS32]	2018	✓	✓	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	✓	-	-	-	-
[PS1]	2018	✓	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-	-	-
[PS18]	2018	✓	✓	-	-	-	✓	-	✓	-	-	-	-	-	-	-	✓	-	-	-	-	-
[PS35]	2018	✓	✓	-	-	-	-	-	-	-	-	✓	-	-	✓	✓	-	-	-	✓	-	✓
[PS9]	2018	✓	✓	-	-	-	-	-	-	-	-	✓	-	-	✓	✓	-	-	-	✓	✓	✓
[PS5]	2018	✓	✓	-	-	-	-	-	-	-	-	✓	-	-	✓	✓	-	-	-	✓	✓	-
[PS2]	2018	-	-	-	-	-	✓	✓	-	-	-	✓	✓	-	-	-	-	-	-	-	✓	✓
[PS25]	2018	✓	✓	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-	-	-	✓	✓
[PS31]	2018	✓	✓	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	✓	-	-	-	-
[PS20]	2019	-	-	✓	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-	✓	✓	✓	✓
[PS26]	2019	✓	✓	✓	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	✓	✓	✓
[PS27]	2019	-	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	✓
[PS3]	2019	✓	-	✓	-	-	-	-	-	✓	-	✓	-	-	✓	✓	-	-	-	✓	-	✓
[PS34]	2019	✓	✓	-	-	-	-	-	-	-	-	-	-	-	✓	-	✓	-	-	-	✓	-
Total		23	26	6	1	5	18	1	4	13	2	28	4	1	15	8	6	3	5	11	16	17

Table 7: List of the most prominent authors.

Author	NS
Marijan, Dusica	8
Liaaen, Marius	6
Jiang, Bo	5
Chan, Wing Kwong	4
Gotlieb, Arnaud	4
Sen, Sagar	3
Tse, T. H.	3
Rothermel, Gregg	3

Table 8: List of countries.

Country	NS
China	9
Norway	9
United States of America	5
Sweden	4
Republic of Korea	3
Australia	2
Canada	2
Finland	1
Germany	1
Pakistan	1
Switzerland	1
United Kingdom	1

4.2 Characteristics of the approaches

In this section, we address the second group of questions concerning the characteristics of the approaches, answering RQ2.1 and RQ2.2, respectively, in Sections 4.2.1 and 4.2.2.

4.2.1 RQ2.1 - Types of approaches. To answer this question, we first identified the research goal of the found studies, according to the dimensions of our schema. The results can be viewed in the first columns of Table 6.

We can observe, most studies (26 out of 35, $\approx 74\%$) compare prioritization methods, followed by the introduction of a prioritization technique (23, $\approx 66\%$). 18 ($\approx 51\%$) studies consider time constraints, 6 ($\approx 17\%$) combine prioritization and selection methods, 5 ($\approx 14\%$) use prioritization for fault localization, and we found 1 study that is included in the category Others and another one combining prioritization with minimization techniques. We did not find studies proposing, explicitly, new evaluation measures.

Some papers do not have a simple goal. For instance, some studies introduce a new technique and also compare it with existing ones. To provide better visualization of this intersection, we generated Figure 5 that presents the interaction between goals and number of studies associated with each one¹. In the figure, the bars represent the number of studies; each category is associated with one or more bars, and the bars are associated with one or more categories. When the bar is associated with more than one category, this means that there is an intersection between them, represented by a line with bullets. Consider the category Comparison (last row). There are 26 studies belonging to this category (horizontal bar in the left). The intersection with the other categories are represented in the right by bullets, 9 studies have intersection only with the category Prioritization; 8 studies with the categories Prioritization and Time-Limit; and so on.

¹The figure was build using UpSet [24].

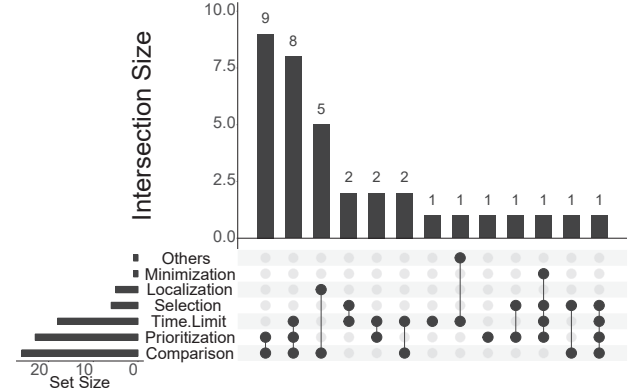


Figure 5: Interaction between research goals.

As evidenced by Figure 5, we observe a large number of studies comparing prioritization techniques. Besides that, we observe that all studies that aim fault localization (5 studies in the third bar) also compare techniques but without any other intersection; they do not propose any new technique, for instance. This can be considered a gap to be explored.

Regarding the information source considered by the approach and our schema (Table 5), we can see in Table 6 that we did not find works on the categories Requirement-based and Model-based. The great majority of the studies (28 out of 35 (80%)) are History-based, followed by the categories Others and Coverage-based with, respectively, $\approx 43\%$ and $\approx 37\%$ of studies, Cost-aware and Human-based with $\approx 11\%$ each, Distribution-based with $\approx 6\%$, and Probabilistic with only 1 study ($\approx 3\%$).

Tables 9 and 10 show the number of information sources used in the studies, as well as, the most commonly used.

Table 9: Number of information sources used in the prioritization process in TCPPI.

#	Perc.	Primary Studies
1	9%	[PS1,PS8,PS35]
2	29%	[PS6,PS9,PS15,PS16,PS18,PS20,PS22,PS26,PS27,PS32]
3	31%	[PS3,PS5,PS7,PS17,PS21,PS23,PS25,PS28,PS31,PS33,PS34]
4	17%	[PS2,PS4,PS19,PS24,PS29,PS30]
6	14%	[PS10-PS14]

In total, we identified a maximum of six sources used at the same time during the prioritization process and 30 different types of information sources. Besides the information used in the prioritization shown in Table 10, we can mention other kinds of information sources used, such as prioritization order history, severity history, correlation data, dissimilarity, text similarity, and text case description. Moreover, most studies use three sources, and the most

Table 10: Most common information sources used in the prioritization process in TCPCL.

Category	Subcategory	Perc.	Primary Studies
History-based	Failure History	77%	[PS2-PS4,PS6,PS7,PS9,PS15-PS35]
	Execution History	54%	[PS2,PS5,PS7,PS17-PS19,PS21-PS33]
Coverage-based	Test Age	9%	[PS4,PS7,PS17]
	Test Coverage	14%	[PS5,PS19,PS23,PS24,PS33]
	Functions not yet covered	14%	[PS10-PS14]
	Functions covered	14%	[PS10-PS14]
	Statements covered	14%	[PS10-PS14]
	Statements not yet covered	14%	[PS10-PS14]
	Branches not yet covered	9%	[PS10,PS12,PS13]
	Branch covered	9%	[PS10,PS12,PS13]
Human-based	Importance	9%	[PS2,PS29,PS30]
Others	Code Changing	9%	[PS16,PS29,PS30]

used kinds of information sources are failure and execution history.

The works in the category History-based consider information about which test cases failed previously. We observed that 19 of them (out of 28, 54%) also consider the Test Execution History, that is, the time to execute the test cases.

Studies in the category Coverage-based use the total number of covered (or not covered) statements, functions, methods, and features [PS3-PS5,PS10-PS14,PS19,PS20,PS23,PS24,PS33].

Works in the category Others use different kinds of information to prioritize the test cases. We identified 8 works that use Search-based or Machine Learning (ML) techniques [PS3-PS5,PS10-PS14,PS28-PS30,PS33,PS34]. Besides that, studies in such a category perform the prioritization considering test case dissimilarity (Diversity-based) [PS9] or test selection strategies [PS8].

In the category Cost-aware [PS5,PS18,PS19,PS21] the studies assume that each test case does not have the same cost, that is, some may be more costly to execute than others, maybe due to the fault severity or running time. One way to evaluate this assumption is by using the APFDc measure proposed by Elbaum et al. [7]. The test cases are usually prioritized until a maximum cost is reached that is feasible to execute.

Works in the category Distribution-based [PS11,PS14] prioritize test cases considering the distribution of their execution profiles via test case distances based on the dissimilarity metrics: count metric and the proportional binary metric. Then, test cases are clustered according to their similarities, allowing, in this way, the identification of redundant test cases and isolating clusters that may cause failures.

We identified only 4 studies [PS2,PS21,PS29,PS30] in the category Human-based. The work of Alegroth et al. [PS2] uses a feature priority ranked by stakeholders, whilst Strandberg et al. [PS29,PS30] use a level of priority defined in each test case by the developers. On the other hand, in the study of Marijan et al. [PS21], a human defines

domain-specific heuristics (weights) for the prioritization according to the organization settings. Consequently, we classified this approach as a prioritization based on user preference.

In the category Probabilistic, Abdullah et al. [PS1] propose an idea of a framework to test each Internet of Things (IoT) layer in a separate Test Server (used as a CI environment). The goal is to prioritize tests based on the frequency of the features derived within an operational profile (which characterizes how a system will be used in production). Then, to predict fault location in IoT services, operational profiles (derived from interface behaviors of IoT services) are combined with Markov chain usage models. This kind of test is also known as usage-based testing.

As we can observe from the description above, most studies consider different kinds of information in the test case prioritization and belong to more than one category. This is illustrated in Figure 6 which presents interactions among the categories and the number of studies associated to each one. Consider the category History-based (last row). There are 28 studies belonging to this category (horizontal bar in the left). The intersection with the other categories are represented in the right by bullets, 12 studies do not have any intersection with any other category (first vertical bar), 3 studies of this category also are included in the Coverage-based category (third bar), 3 in the category Others, and so on.

As mentioned before and evidenced by Figure 6, we observe a large number of approaches based on historical information about test cases previously executed in CI environments. To obtain other kinds of information can be not possible due to the constraints in the CI environments. As a consequence, there is an effort to create lightweight approaches. Approaches that require exhaustive analysis are costly and inefficient [PS7], as well as the time available to run the prioritized test suite can be reduced if prioritization takes too long [14].

To better analyze this fact, we can see in Figure 7 the categories used over the years. Coverage-based, Distribution-based, and Others were the first kind of

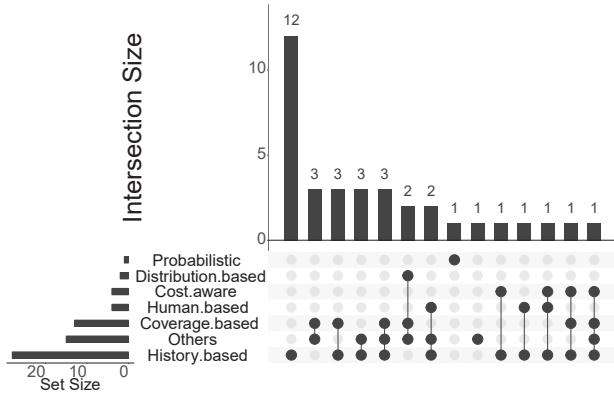


Figure 6: Intersection between the categories regarding considered information.

categories used. The use of historical test data appeared in 2011. Since then, we observe a growing number of studies in this category, mainly in the last two years 2017 and 2018, when 14 studies in the category History-based (out of 28, 50%) were published. In this way, we can observe a growing number of studies in this category and a trend in the field.

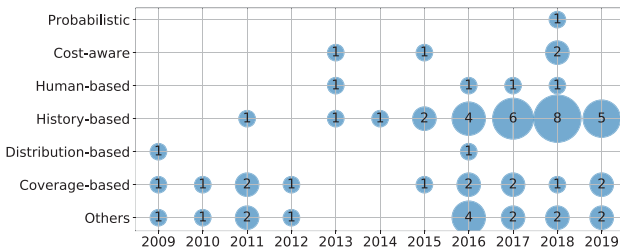


Figure 7: Categories regarding information used over the years.

4.2.2 RQ2.2 - Application contexts. To characterize the application context of the found approaches, we collected information about the programming language, requirements, environments explored, and limitations, as well as some CI particularities and testing problems.

We observed that few studies (11 out of 35, 31%) make explicit the programming language addressed. From them, we found only three programming languages: C in six studies (55%) [PS5,PS10-PS14], Java in five studies (45%) [PS3,PS5,PS6,PS9,PS15], and Ruby in one study (9%) [PS18]. Many studies use systems (datasets) from companies. Consequently, details about the used systems are limited. Furthermore, some datasets are a sample of products including many programming languages, i.e., Google Shared Dataset of Test Suite Results (GSDTSR) [8].

On the other hand, we observed that 31% of the studies [PS3-PS5,PS9-PS14,PS16,PS33] need code analysis to perform the prioritization. Most of these studies [PS3-PS5,PS10-PS14,PS33] belong to the Coverage-based category, whilst the other ones analyse code changing [PS16] and dissimilarity of the test cases [PS9].

In the Coverage-based category only 4 studies do not require code analysis [PS19,PS20,PS23,PS24]. Three studies [PS20,PS23,PS24] investigate TPCPI in Highly Configurable Systems (HCS) context, and one study evaluates three Android systems [PS19]. Such studies analyze feature coverage.

In relation to the CI environments investigated, we identified ten industrial environments and two CI frameworks (services). The most common CI environment investigated is from Google [PS7,PS17,PS18,PS27,PS28,PS32,PS33,PS35], once that the GSDTSR dataset from Google is available online² allowing easy use and comparison. This environment is followed by the Cisco environment [PS20-PS25,PS27], result of the cooperation between Simula Research Laboratory and Cisco company. ABB Robotics appears in the sequence [PS27,PS28,PS31] and it also has its datasets IOF/ROL and Paint Control available online³. The other CI environments are: Ericsson [PS16,PS26], Westermo [PS29,PS30], Axis Communications [PS16], Baidu [PS5], LexisNexis [PS34], Salesforce.com [PS4], and Techship [PS2].

Representing the CI frameworks category we found Jenkins [PS19] and Travis CI [PS9,PS18]. Travis CI is the most popular CI framework in the GitHub, accounting for roughly 50% of the CI market⁴, followed by Circle CI, and Jenkins. The use of CI frameworks can be considered a gap to be explored.

Table 11 presents the CI problems and particularities investigated/addressed by the studies.

We identified that 34% of the studies (12 out of 35) do not explicitly mention some CI testing problem addressed. Most studies (18 out of 26, around 78%) deal with Time-Consuming testing (or Time-Limit Treatment), once that this restriction is easier to deal rather than other problems.

43% (10 out of 26) of the studies address the Complex testing problem. In such a problem, HCS are the most investigated (7 studies), followed by Android and IoT systems (1 study each). The CI testing problems Flaky tests and UI testing are few explored, respectively, 13% and 9% of the studies address such problems. To investigate such CI testing problems is a gap for future work.

²The GSDTSR dataset can found at <https://code.google.com/archive/p/google-shared-dataset-of-test-suite-results/>.

³The datasets from ABB Robotics can found at <https://bitbucket.org/HelgeS/atcs-data>.

⁴Information extracted from blog: [GitHub welcomes all CI tools](#).

Table 11: Continuous Integration problems and particularities in TCPCI field.

Description	Perc.	Primary Studies
<i>Continuous Integration Testing Problems</i>		
User Interface (UI) testing	9%	[PS2,PS34]
Time-Consuming testing	78%	[PS2,PS8,PS18–PS33]
Flaky tests	13%	[PS4,PS7,PS33]
Complex testing	43%	[PS1,PS19–PS25,PS27,PS35]
<i>Continuous Integration Particularities</i>		
Parallelism	14%	[PS1,PS7,PS17,PS27,PS35]
Volatility	34%	[PS3,PS4,PS7,PS9,PS17,PS18,PS23,PS28–PS31,PS35]
Multiple Commits (Test Requests)	11%	[PS7,PS18,PS26,PS35]

Regarding the CI particularities, we observed that 14% of the studies consider Parallelism, that is a particularity of the CI environment where multiple test machines are used and test cases are executed in parallel. Even with modern large-scale parallel test infrastructures, it can take a relatively long time until an engineer has received complete testing feedback for a given change [PS4].

Another particularity of the CI environment is the Volatility of the test cases, that is, that new test cases can be added or removed (discontinued) during the software life-cycle. Although the studies aim to address TCPCI, we identified few studies (12 out 35, $\approx 34\%$), which explicitly consider this particularity.

Only 11% of the studies consider Multiple Commits (or multiple test requests). This particularity is related to the fact that the CI environment can receive multiple requests (commits) to test at the same time, and an order to test them is required. We observed that only two studies consider prioritization of commits [PS18,PS35]. To address this particularity is a research opportunity.

We also examined the works considering another particularity about the use of resources in CI environments, such as memory consumption and the use of GPU (Graphics Processing Unit) to allow a fast prioritization process. However, we have not identified any related work.

We can see that many approaches search for code independence, due to the time constraints of a CI environment, as well as independence regarding programming language. However, few studies address CI environment particularities and problems, such as parallel execution of test suites, the impact of time constraints, and volatility of test cases.

4.3 Evaluation Aspects

The evaluation phase is especially important to check the usefulness and applicability of the proposed approaches. In this section, we provide answers for RQ3, by analysing how the approaches have been evaluated (RQ3.1, Section 4.3.1) and how the results are measured and whether they are statistically analyzed (RQ3.2, Section 4.3.2).

4.3.1 RQ3.1 - Evaluation of the approaches. It is interesting to observe that almost all the found studies report evaluation results, except two studies [PS1,PS8] that are theoretical. A possible reason for this is the fact that prioritization of test cases is related to a practical issue. Then, 33 studies were considered in our analysis to characterize the evaluation contexts and answer our research questions.

We identified 99 systems (or datasets) used in the studies. Most of them were few used, $\approx 81\%$ of them were used once, and $\approx 9\%$ twice. In this sense, we analyzed the systems used more than twice (10%). Among them, GS-DTSR [8] was used seven times. The other systems used more than twice are Video Conferencing Systems from Cisco, Paint Control from ABB Robotics, and the systems that are part of Siemens suites obtained from the Software-artifact Infrastructure Repository (SIR)⁵: tot_info, schedule, print_tokens2, replace, print_tokens, schedule2, and tcas.

A rigorous evaluation methodology should consider the techniques selected for a comparison with the proposed approach. In this sense, we identified 59 techniques commonly used as a baseline. The most common is Random TCP used in 19 studies out of 33 ($\approx 54\%$), followed by the ROCKET technique [PS21] and the use of untreated tests (that is the use of the same order of the original set), both used in 18% of the studies. We can also highlight the Manual prioritization, used as a baseline in $\approx 15\%$ of the studies.

We also analyzed the main threats mentioned by the authors in the studies. 11 studies [PS1,PS8,PS11,PS16,PS23,PS24,PS29–PS33] do not mention any threat. Among the studies that mention threats, the most common one that appeared in 100% of the studies is related to the Systems used. This fact points out some concerns about the scalability and generalization of the approaches proposed. After, threats in the categories *Evaluation Measure* (58%), *Techniques Compared* and *Experimental Settings* appear with similar frequency (38% each). Other threats are related to the tool used, randomness aspects, available resources, etc.

To understand how these threats can be related to CI testing problems (discussed in Section 4.2.2) and possibly affect them, we build Figure 8. In such a figure, we provide an illustration showing the relationship between the dimensions: CI testing problems, CI particularities, and the main limitations (threats) found in the studies.

We can also observe the maturity of the TCPCI field regarding the investigation of CI testing problems and the

⁵Available at <http://sir.unl.edu/>

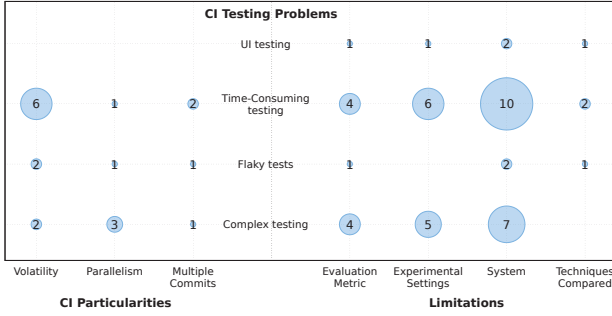


Figure 8: Relationship of the studies concerning CI particularities, CI testing problems, and the main limitations (threats) found by authors.

most involved limitations. For instance, most studies that consider the CI particularity, Volatility of test cases, also seek to address the Time-Consuming testing problem. Still, they have the main limitation concerning the systems used in the experiments. Moreover, the UI testing problem has not been investigated considering any CI particularity, and this should be explored in future work.

The figure shows that there are more limitations than solutions in the TCPCI field. In this sense, there is a lack of studies that consider, at the same time, CI particularities and CI problems.

4.3.2 RQ3.2 - Evaluation Measures. Evaluation measures play an essential role in measuring the efficacy of an approach, as well as to benchmark its effectiveness against other existing ones. In this sense, we identified 23 evaluation measures. Among them, 6 are the most common (used), and the other ones were grouped in the category Others.

As we can see in Table 6 the most widely used measure is Time, used in 16 out of 33 studies (48%), followed by FD (33%), APFD (24%), APFDc (18%), Expense (15%), and NAPFD (9%). In the studies, the measure Time can encompass: test case execution time, the time the approach takes to perform the prioritization, and the time reduction obtained. In this way, we can use *Time* concerning the execution time of an approach and test case execution time to measure the efficiency of the approach, whilst to measure the effectiveness we can use the time reduction obtained. This last one concerns the time spent to the first failure, which can impact the time spent in a CI cycle and consequently, the test cost.

Average Percentage of Faults Detected (APFD) [30] indicates how quickly a set of prioritized test cases (T') can detect faults present in the application being tested, and its value is calculated from the weighted average of the percentage of detected faults. Higher APFD values indicate that the faults are detected faster using fewer test cases.

Number or percentage of Faults Detected (FD) is a simple version of the APFD measure, which considers only the fault detected rate. This measure can be used to compare the faults detected in the test reduction, for instance, concerning a time constraint or a percentage of test case executed. Besides that, we can find in the literature [PS20] a formalization of this measure as *Fault-detection effectiveness*. This measure considers the ratio of the number of *nonrepeated* faults detected by the reduced test suite and the number of *nonrepeated* faults detected by its original (nonreduced) test suite. The values are between 0 and 1, higher values indicate better fault-detection effectiveness of a test suite. *Nonrepeated* faults are faults counted only once regardless of the number of test cases that detect that fault. However, to identify *nonrepeated* faults can be a hard task.

Expense [16, 18, 39] is a common metric to measure the effectiveness of fault localization, and it is computed by dividing the number of statements needed to be examined to find a specific fault by the total number of executable statements in the program. A technique with a smaller expense to locate a particular fault means better fault localization effectiveness.

APFDc (APFD with cost consideration) [7] and NAPFD (Normalized APFD) [29] [PS28] are measures adapted from APFD. APFDc was proposed to deal with an APFD limitation concerning the assumption that all faults have equal severity and the test cases have equal costs. These assumptions are not possible in practice, and therefore, APFDc takes into account the fault severity and test cost. Furthermore, if both fault severity and test case costs are identical, APFDc can be used to compute the APFD value. The NAPFD measure is an extension of APFD. To calculate NAPFD, we consider the ratio between detected and detectable faults within T . This measure is adequate to prioritize test cases when not all of them are executed, and faults can be undetected.

Besides the most common evaluation measures described before, we identified 17 measures which were grouped in the category Others, such as recall, precision, f-measure, the area under a curve, number of test cases run, the test case/suite size, requirement coverage, and successful fault localization percentage.

Although many studies conducted experiments and analysis of results, few of them (10 out 33 studies $\approx 30\%$) apply a statistical test. Table 12 presents the statistical tests used in the primary studies. Among them, ANOVA is the most preferred test. There is less use concerning the pairwise comparison and effect size measurement. Through this analysis, we observed a lack of statistical tests applied to evaluate the results.

Table 12: Statistical tests applied in TCPCI field.

Statistical Tests	Percentage	Primary Studies
Pairwise Comparison		
Wilcoxon Mann-Whitney	9%	[PS9,PS20,PS27]
Multiple Comparison		
ANOVA	18%	[PS5,PS10-PS14]
Dunn's	3%	[PS27]
Kruskal-Wallis	3%	[PS27]
Scott-Knott analysis	3%	[PS34]
Tukey's HSD	6%	[PS5,PS13]
Effect Size Measurement		
Cliff's Delta	3%	[PS34]
Vargha-Delaney \hat{A} measure	6%	[PS9,PS27]

5 TRENDS AND RESEARCH OPPORTUNITIES

During the analysis of the found studies, we identified some trends and research gaps. Based on them, we discuss in this section, the main research opportunities related to each research question. We also present some trends in the field.

5.1 Evolution of the field

By analysing the publications over the years, we can conclude that TCP is an emergent research topic. The great majority of the found studies (21 out of 35, 60%) have been published in the last three years (2016-2018). As a consequence, we observe that there are few research groups working on this subject, and there is space for innovation.

5.2 Type of approaches

As mentioned in Section 4.2.1, we observe a trend and preference by History-based techniques. Coverage-based techniques and search-based ones can take time to perform the prioritization. The fault-based history is the most used, but there are other kinds of information sources to be taken into account, such as execution time, relation with non-functional requirements, and so on. Moreover, we have not found studies addressing Model-based prioritization, considering, for example, behavior models and UML diagrams.

As evidenced by Table 10, most approaches use two and three kinds of information sources. In this path, it is necessary to investigate the impact of the amount of information and identify the most reliable kind of information in the prioritization process. Another research direction observed is exploiting different information sources during the prioritization process. Such a consideration falls in the use of techniques which deal properly with multi-information and dimensionality problem, a gap for future research.

We observed a trend that is to explore Artificial Intelligence (AI) techniques like Machine Learning and probabilistic ones, as well as the use of search-based algorithms with a focus on multi- and many-objectives. According to Spieker et al. [PS28] and Chen et al. [PS5], the use of Deep Learning techniques can be a promising path for future research in the TCPCI context. Moreover, search-based techniques can be time-consuming concerning the time spent to find a suitable solution. Besides that, these techniques need to consider the available resources, such as memory consumption. In this sense, we observed that the use of GPU could be considered (not explored in the TCPCI context), as well as the use of Blockchain with Machine Learning⁶ which is considered a trend in the AI field.

In the Search-Based Software Engineering (SBSE) field, TCP approaches exploring user preferences to guide the search for the best test case order [10] have been proposed recently. The use of such preferences has been explored in a few studies (only 4 studies in the Human-based category). Another trend in SBSE is the use of Hyper-Heuristics [9] that can provide flexible and adaptable solutions for testing problems. These characteristics can be useful, considering the TCPCI dynamic environment.

Another research opportunity is to explore the use of user requirements and organizational constraints to prioritize the test cases. Furthermore, we identified a lack of studies proposing new techniques for TCP in combination with fault localization. This is a gap to be explored by future research that should consider the CI environment characteristics and challenges. According to Jiang et al. [PS14], a gap is to study how to achieve tighter integration between regression testing and debugging techniques. Additionally, Yu et al. [PS34] mention that the use of fault location might help to address the test failure classification problem. In such a problem, each test failure can be related to a specific fault, caused by some piece of code.

5.3 Application contexts

We identified that only three programming languages were addressed by the studies: Java, C, and Ruby. The use of TCPCI for other popular programming languages, such as C# and Python, needs to be explored [1].

We also identified only ten industrial environments and only two CI Frameworks: Travis CI and Jenkins. Considering other industrial scenarios with different kinds of systems is a gap. From such environments, only two environments made available their data used in their research: Google with the GSDTSR dataset and ABB Robotics with its datasets IOF/ROL and Paint Control. Moreover, only two studies made available the proposed approach [PS5,PS28]. This hampers the replication of the experiments and undermines a more open science. We suggest future researches

⁶Microsoft Research Blog: [Leveraging blockchain to make machine learning models more accessible](#).

to use [Open Science Framework](#) to increase the openness, integrity, accessibility, and reproducibility of scientific research, consequently contributing to the Open Science community.

We observed that among the CI testing problems, UI testing and Flaky tests have been few addressed. The most addressed problem is related to time constraints. Concerning complex testing, a gap is to investigate TCPCI techniques in emergent, new contexts, and systems that are hard to test, i.e., UI systems, Service-based systems, dynamic applications, apps phones, and HCS.

We also found few works addressing some CI particularities such as parallelism in the environment, the volatility of test cases, and multiple commits. Future works should focus on CI specific characteristics, such as the use of the available resources, for instance, memory consuming.

5.4 Evaluation of the approaches

Almost all studies (33 out of 35) are dedicated to compare and evaluate the proposed approaches. Recent studies are concerned with the approach effectiveness evaluation by using measures such as Time required to perform the prioritization, given the CI time constraints, but FD is also a measure used by a great number of studies. A gap is the use of NAPFD measure, few explored.

Few studies use a statistical test and are worried about scalability. In this way, more rigorous experiments are necessary to evaluate these aspects and perform a comparison of the approaches in practice. We identified a list of 99 systems used in TCPCI context (see Section 4.3.1, RQ3.1). However, a lack of a benchmark for the field is a gap to be addressed.

GSDTSR is the most used system which contains test suite results from a sample of Google, and it could be used in future studies. As mentioned before, Travis CI is the most popular CI framework in the GitHub, as well as it provides an API to access test results from open-source repositories. A data mining procedure could be conducted to identify different systems, such as HCS and Android. For this, the use of Travis Torrent could be considered [2]. A possible research opportunity is the construction of a repository containing a set of systems that are considered deemed TCP cases and have different characteristics regarding the number of faults by commit and the number of test cases. This could help to overcome the main threats found in the evaluations and reported in the studies.

There is a lack of studies addressing CI particularities and testing problems at the same time. This hampers the evolution of the TCPCI field, mainly due to the high number of limitations identified. The creation of a benchmark, the use of adequate evaluation measures, and the availability of techniques can accelerate research in the TCPCI context.

Moreover, a qualitative analysis of the impact of the techniques in industrial settings might be considered as future research.

6 THREATS TO VALIDITY

In this section, we identify possible threats to the validity of our results, according to the taxonomy of Wohlin et al. [37].

Regarding construct validity, the research questions may not address all TCPCI aspects. This threat was mitigated through discussions, and we believe that the questions reflect the goals of our work. Other authors can elaborate other questions and obtain a different analysis. The databases used are well-known and related to the software engineering and software testing areas. In this way, we believe the found studies represent well the TCPCI field. Our search string does not have many elements, but we carefully created a search string capable of finding consistent results. To construct the search string, we selected terms related to our goals and synonymous used in mappings of the TCP field [4, 20]. We refined our search string several times by using a control group, reducing the risk that relevant literature is omitted.

Concerning internal validity, maybe we may have extracted and misinterpreted some information. To minimize such a threat during the data extraction, we followed a rigorous plan, using the PRISMA statement, and well-defined inclusion and exclusion criteria. We had meetings and discussions to clarify any doubt arisen during the process.

Other possible threats, related to the conclusion validity, is the granularity of the information described in the studies, which may affect our conclusions. Besides, our schema can also be a threat, as well as the form we classified the papers. To mitigate them, we first documented all relevant information from the primary studies guided by the dimensions associated with the research questions and defined the categories interactively. However, other researchers may obtain another scheme and ways to group and analyze the papers.

Regarding reliability validity, our study can be easily replicated, following the steps described and using the search string or using the raw data analyzed and disseminated by the Open Science Framework (OSF).

7 CONCLUDING REMARKS

In this paper, we present the results of a systematic mapping study on the TCPCI field. We investigated some aspects of the found studies: the main research goal, characteristics of the explored approach, used evaluation measures, and how the evaluation has been conducted. Furthermore, we also analyzed the main publication fora and how the field has been evolved over the years, trends, and research opportunities to guide future research.

The map found 35 papers, published in a wide range of venues, without the indication of a preferred publication

vehicle. We observed an increasing number of studies in the last years and a crescent interest in the field.

The main research goal of the studies is the comparison of prioritization techniques followed by the introduction of a prioritization technique. Concerning the information considered in the prioritization, History-based approaches seem to be a trend. Probabilistic and Distribution-based approaches were explored by only one study. To explore new sources of information to prioritize the test cases such as requirements, organizational restrictions, and human aspects can be an alternative, maybe in combination with historical information. We identified few studies combining TCP and fault localization, and there is no approach based on requirements or models. These are gaps that can guide future research.

To evaluate the proposed approaches, several systems are used. We identified 99 different systems. GSDTSR is the most common system used. The main threat found in the evaluations is concerned with the systems used, followed by the evaluation measures and techniques used in the comparison. We observed that only 9 studies apply a statistical test. Regarding evaluation measures, we identified 23 measures in the primary studies, and Time and FD are the most used.

Some research opportunities are the application of TCPCI in other contexts considering languages such as C# and Python. It is necessary to address other CI testing problems and particularities such as complex and time-consuming testing, flaky tests, UI testing, parallelism, test case volatility, and multiple commits. Other limitations to be overcome are the use of different systems with different characteristics (size, faults by commit, number of commits, number of test cases) to allow generalization and scalability evaluation, as well as the construction of benchmarks.

ACKNOWLEDGMENTS

This work is supported by the Brazilian agencies CAPES and CNPq. (Grant: 305968/2018-1).

PRIMARY STUDIES

- [PS1] Ahmed Abdullah, Heinz W. Schmidt, Maria Spichkova, and Huai Liu. 2018. Monitoring Informed Testing for IoT. In *Proceedings of the 25th Australasian Software Engineering Conference (ASWEC)*. IEEE, 91–95. <https://doi.org/10.1109/ASWEC.2018.00020>
- [PS2] Emil Alégroth, Arvid Karlsson, and Alexander Radway. 2018. Continuous Integration and Visual GUI Testing: Benefits and Drawbacks in Industrial Practice. In *Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 172–181. <https://doi.org/10.1109/ICST.2018.00026>
- [PS3] Sadia Ali, Yaser Hafeez, Shariq Hussain, and Shunkun Yang. 2019. Enhanced regression testing technique for agile software development and continuous integration strategies. *Software Quality Journal* (Sep 2019). <https://doi.org/10.1007/s11219-019-09463-4>
- [PS4] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 975–980. <https://doi.org/10.1145/2950290.2983954>
- [PS5] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing Test Prioritization via Test Distribution Analysis. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 656–667. <https://doi.org/10.1145/3236024.3236053>
- [PS6] Younghwan Cho, Jeongho Kim, and Eunseok Lee. 2016. History-Based Test Case Prioritization for Failure Information. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 385–388. <https://doi.org/10.1109/APSEC.2016.066>
- [PS7] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, New York, NY, USA, 235–245. <https://doi.org/10.1145/2635868.2635910>
- [PS8] Martin Eyl, Clements Reichmann, and Klaus Müller-Glaser. 2016. Fast Feedback from Automated Tests Executed with the Product Build. In *Proceedings of the 8th International Conference on Software Quality (SWQD)*. Springer, 199–210. https://doi.org/10.1007/978-3-319-27033-3_14
- [PS9] Alireza Haghighatkah, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98. <https://doi.org/10.1016/j.jss.2018.08.061>
- [PS10] Bo Jiang and Wing Kwong Chan. 2010. On the Integration of Test Adequacy, Test Case Prioritization, and Statistical Fault Localization. In *Proceedings of the 10th International Conference on Quality Software (QSIC)*. IEEE, 377–384. <https://doi.org/10.1109/QSIC.2010.64>
- [PS11] Bo Jiang and Wing Kwong Chan. 2016. Testing and Debugging in Continuous Integration with Budget Quotas on Test Executions. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 439–447. <https://doi.org/10.1109/QRS.2016.66>
- [PS12] Bo Jiang, Wing Kwong Chan, and T. H. Tse. 2011. On Practical Adequate Test Suites for Integrated Test Case Prioritization and Fault Localization. In *Proceedings of the 11th International Conference on Quality Software (QSIC)*. IEEE, 21–30. <https://doi.org/10.1109/QSIC.2011.37>
- [PS13] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, T. H. Tse, and Tsong Yueh Chen. 2012. How Well Does Test Case Prioritization Integrate with Statistical Fault Localization? *Information and Software Technology* 54, 7 (July 2012), 739–758. <https://doi.org/10.1016/j.infsof.2012.01.006>
- [PS14] Bo Jiang, Zhenyu Zhang, T. H. Tse, and Tsong Yueh Chen. 2009. How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 99–106. <https://doi.org/10.1109/COMPSAC.2009.23>
- [PS15] Jeongho Kim, Hohyeon Jeong, and Eunseok Lee. 2017. Failure History Data-based Test Case Prioritization for Effective Regression Test. In *Proceedings of the Symposium on Applied Computing (SAC)*. ACM, New York, NY, USA, 1409–1415. <https://doi.org/10.1145/3019612.3019831>
- [PS16] Eric Knauss, Mirosław Staron, Wilhelm Meding, Ola Söder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-Churn Based Test Selection. In *Proceedings of the IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering (RCoSE)*. IEEE, 19–25. <https://doi.org/10.1109/RCoSE.2015.11>
- [PS17] Jung-Hyun Kwon and In-Young Ko. 2017. Cost-Effective Regression Testing Using Bloom Filters in Continuous Integration Development Environments. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 160–168. <https://doi.org/10.1109/APSEC.2017.22>
- [PS18] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining Prioritization: Continuous Prioritization for Continuous Integration. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 688–698. <https://doi.org/10.1145/3180155.3180213>
- [PS19] Dusica Marijan. 2015. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE Computer Society, 157–162. <https://doi.org/10.1109/QRS.2015.31>
- [PS20] Dusica Marijan, Arnaud Gotlieb, and Marius Liaaen. 2019. A learning algorithm for optimizing continuous integration development and

- testing practice. *Software: Practice and Experience* 49, 2 (2019), 192–213. <https://doi.org/10.1002/spe.2661>
- [PS21] Dusica Marijan, Arnaud Gottlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICMS)*. IEEE, 540–543. <https://doi.org/10.1109/ICSM.2013.91>
- [PS22] Dusica Marijan and Marius Liaaen. 2016. Effect of Time Window on the Performance of Continuous Regression Testing. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 568–571. <https://doi.org/10.1109/ICSME.2016.77>
- [PS23] Dusica Marijan and Marius Liaaen. 2017. Test Prioritization with Optimally Balanced Configuration Coverage. In *Proceedings of the IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 100–103. <https://doi.org/10.1109/HASE.2017.26>
- [PS24] Dusica Marijan, Marius Liaaen, Arnaud Gottlieb, Sagar Sen, and Carlo Ieva. 2017. TITAN: Test Suite Optimization for Highly Configurable Software. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 524–531. <https://doi.org/10.1109/ICST.2017.60>
- [PS25] Dusica Marijan, Marius Liaaen, and Sagar Sen. 2018. DevOps Improvements for Reduced Cycle Times with Integrated Test Optimizations for Continuous Integration. In *Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. IEEE, 22–27. <https://doi.org/10.1109/COMPSAC.2018.00012>
- [PS26] Armin Najafi, Weiye Shang, and Peter C. Rigby. 2019. Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Press, Piscataway, NJ, USA, 213–222. <https://doi.org/10.1109/ICSE-SEIP.2019.00031>
- [PS27] Dipesh Pradhan, Shuai Wang, Shaikat Ali, Tao Yue, and Marius Liaaen. 2019. Employing rule mining and multi-objective search for dynamic test case prioritization. *Journal of Systems and Software* 153 (2019), 86–104. <https://doi.org/10.1016/j.jss.2019.03.064>
- [PS28] Helge Spieker, Arnaud Gottlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, 12–22. <https://doi.org/10.1145/3092703.3092709>
- [PS29] Per Erik Strandberg, Wasif Afzal, Thomas J. Ostrand, Elaine J. Weyuker, and Daniel Sundmark. 2017. Automated System-Level Regression Test Prioritization in a Nutshell. *IEEE Software* 34, 4 (July 2017), 30–37. <https://doi.org/10.1109/MS.2017.92>
- [PS30] Per Erik Strandberg, Daniel Sundmark, Wasif Afzal, Thomas J. Ostrand, and Elaine J. Weyuker. 2016. Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors. In *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 12–23. <https://doi.org/10.1109/ISSRE.2016.23>
- [PS31] Wen Wen, Zhongju Yuan, and Yuyu Yuan. 2018. Improving RETECS method using FP-Growth in continuous integration. In *Proceedings of the 5th IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*. IEEE, 636–639. <https://doi.org/10.1109/CCIS.2018.8691385>
- [PS32] Lei Xiao, Huaikou Miao, and Ying Zhong. 2018. Test case prioritization and selection technique in continuous integration development environments: a case study. *International Journal of Engineering & Technology* 7, 2.28 (2018), 332–336. <https://doi.org/10.14419/ijet.v7i2.28.13207>
- [PS33] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster Fault Finding at Google Using Multi Objective Regression Test Optimisation. In *Proceedings of the 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’11)*. Industry Track.
- [PS34] Zhe Yu, Fahmid Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian. 2019. TERMINATOR: better automated UI test case prioritization. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. ACM, 883–894. <https://doi.org/10.1145/3338906.3340448>
- [PS35] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. 2018. Test Reprioritization in Continuous Testing Environments. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 69–79. <https://doi.org/10.1109/ICSME.2018.00016>

REFERENCES

- [1] [n.d.]. The RedMonk Programming Language Rankings: January 2018s. <https://redmonk.com/sograzy/2018/03/07/language-rankings-1-18/>. Accessed: 2018-03-20.
- [2] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.
- [3] Buildbot. [n.d.]. The Continuous Integration Framework. <https://buildbot.net>. Accessed: 2018-01-22.
- [4] Cagatay Catal and Deepti Mishra. 2013. Test Case Prioritization: A Systematic Mapping Study. *Software Quality Journal* 21, 3 (09 2013), 445–478. <https://doi.org/10.1007/s11219-012-9181-z>
- [5] Theodore D. Hellmann, Abhishek Sharma, Jennifer Ferreira, and Frank Maurer. 2012. Agile Testing: Past, Present, and Future – Charting a Systematic Map of Testing in Agile Software Development. In *Proceedings of the Agile Conference*, 55–63. <https://doi.org/10.1109/Agile.2012.8>
- [6] Alexander Eck, Falk Uebernickel, and Walter Brenner. 2014. Fit for continuous integration: How organizations assimilate an agile practice. In *Proceedings of the 20th Americas Conference on Information Systems (AMCIS)*. Association for Information Science.
- [7] S. Elbaum, A. Malishevsky, and G. Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001*. 329–338. <https://doi.org/10.1109/ICSE.2001.919106>
- [8] Sebastian Elbaum, Andrew McLaughlin, and John Penix. 2014. The Google Dataset of Testing Results.
- [9] T. N. Ferreira, J. A. P. Lima, A. Strickler, J. N. Kuk, S. R. Vergilio, and A. Pozo. 2017. Hyper-Heuristic Based Product Selection for Software Product Line Testing. *IEEE Computational Intelligence Magazine* 12, 2 (May 2017), 34–45. <https://doi.org/10.1109/MCI.2017.2670461>
- [10] T. N. Ferreira, S. R. Vergilio, and J. T. de Souza. 2017. Incorporating user preferences in search-based software engineering: A systematic mapping study. *Information and Software Technology* 90 (2017), 55–69. <https://doi.org/10.1016/j.infsof.2017.05.003>
- [11] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2016. The Need for Multivocal Literature Reviews in Software Engineering: Complementing Systematic Literature Reviews with Grey Literature. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, Article 26, 6 pages. <https://doi.org/10.1145/2915970.2916008>
- [12] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), 101 – 121. <https://doi.org/10.1016/j.infsof.2018.09.006>
- [13] GoCD. [n.d.]. Open Source Continuous Delivery and Release Automation Server. <https://www.gocd.org>. Accessed: 2018-01-22.
- [14] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-box and Black-box Test Prioritization. In *Proceedings of the 38th International Conference on Software Engineering (ICSE’16)*. ACM, 523–534. <https://doi.org/10.1145/2884781.2884791>
- [15] Integrity. [n.d.]. Continuous Integration Server. <https://integrity.github.io>. Accessed: 2018-01-22.
- [16] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault Localization Using Value Replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA’08)*. ACM, 167–178. <https://doi.org/10.1145/1390630.1390652>
- [17] Jenkins. [n.d.]. <https://wiki.jenkins-ci.org/display/JENKINS/Home>. Accessed: 2018-01-22.
- [18] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE’05)*. ACM, 273–282. <https://doi.org/10.1145/1101908.1101949>
- [19] Teemu Karvonen, Woubshet Behutiye, Markku Oivo, and Pasi Kuvaja. 2017. Systematic literature review on the impacts of agile release engineering practices. *Information and Software Technology* 86 (2017),

- 87 – 100. <https://doi.org/10.1016/j.infsof.2017.01.009>
- [20] Muhammad Khatibsyaribini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93. <https://doi.org/10.1016/j.infsof.2017.08.014>
- [21] Amit Kumar and Karambir Singh. 2014. A Literature Survey on test case prioritization. *Compusoft* 3, 5 (2014), 793.
- [22] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. 2017. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology* 82 (2017), 55 – 79. <https://doi.org/10.1016/j.infsof.2016.10.001>
- [23] Marco Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. 2015. The highways and country roads to continuous deployment. *IEEE Software* 32, 2 (Mar 2015), 64–72. <https://doi.org/10.1109/MS.2015.50>
- [24] Alexander Lex, Nils Gehlenborg, Hendrik Strobel, Romain Vuillemot, and Hanspeter Pfister. 2014. UpSet: Visualization of Intersecting Sets. *IEEE Transactions on Visualization and Computer Graphics (InfoVis’14)* 20, 12 (2014), 1983–1992.
- [25] Mika V. Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. 2015. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering* 20, 5 (01 Oct 2015), 1384–1425.
- [26] David Moher, Alessandro Liberati, Jennifer Tetzlaff, and Douglas G Altman. 2009. Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement. *BMJ* 339 (2009). <https://doi.org/10.1136/bmj.b2535>
- [27] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. 2008. Systematic Mapping Studies in Software Engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE’08)*. BCS Learning & Development Ltd., 68–77.
- [28] Jackson A Prado Lima and Silvia R Vergilio. 2020. Supplementary Material - Test Case Prioritization in Continuous Integration Environments: A Mapping Study. <https://doi.org/10.17605/OSF.IO/ZFE64>
- [29] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. 2007. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *IEEE International Conference on Software Maintenance*. 255–264. <https://doi.org/10.1109/ICSM.2007.4362638>
- [30] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM’99)*. IEEE Computer Society, 179–188. <https://doi.org/10.1109/ICSM.1999.792604>
- [31] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct. 2001), 929–948. <https://doi.org/10.1109/32.962562>
- [32] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943. <https://doi.org/10.1109/ACCESS.2017.2685629>
- [33] Yogesh Singh, Arvinder Kaur, Bharti Suri, and Shweta Singhal. 2012. Systematic Literature Review on Regression Test Prioritization Techniques. *Informatica* 36, 4 (2012), 379–408.
- [34] Daniel Ståhl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87 (2014), 48 – 59. <https://doi.org/10.1016/j.jss.2013.08.032>
- [35] Travis CI. [n.d.]. Travis CI. <https://travis-ci.org>. Accessed: 2018-01-22.
- [36] C Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE’14)*. ACM, 38:1–38:10. <https://doi.org/10.1145/2601248.2601268>
- [37] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.
- [38] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification & Reliability* 22, 2 (March 2012), 67–120. <https://doi.org/10.1002/stv.430>
- [39] Yanbing Yu, James A. Jones, and Mary Jean Harrold. 2008. An Empirical Study of the Effects of Test-suite Reduction on Fault Localization. In *Proceedings of the 30th International Conference on Software Engineering (ICSE’08)*. ACM, 201–210. <https://doi.org/10.1145/1368088.1368116>
- [40] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: A large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 60–71. <https://doi.org/10.1109/ASE.2017.8115619>

APPENDIX B – A MULTI-ARMED BANDIT APPROACH FOR TCPCI

A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments

Jackson A. Prado Lima
DInf, Federal University of Paraná
Curitiba, Brazil
jackson.lima@ufpr.br

Silvia R. Vergilio
DInf, Federal University of Paraná
Curitiba, Brazil
silvia@inf.ufpr.br

ABSTRACT

Continuous Integration (CI) environments have been increasingly adopted in the industry to allow frequent integration of software changes, making software evolution faster and cost-effective. In such environments, Test Case Prioritization (TCP) techniques play an important role to reduce regression testing costs, establishing a test case execution order that usually maximizes early fault detection. Existing works on TCP in CI environments (TCPCI) present some limitations. Few pieces of work consider CI particularities, such as the test case volatility, that is, they do not consider the dynamic environment of the software life-cycle in which new test cases can be added or removed (discontinued), characteristic related to the Exploration versus Exploitation (EvE) dilemma. To solve such a dilemma an approach needs to balance: i) the diversity of test suite; and ii) the quantity of new test cases and test cases that are error-prone or that comprise high fault-detection capabilities. To deal with this, most approaches use, besides the failure-history, other measures that rely on code instrumentation or require additional information, such as testing coverage. However, to maintain the information updated can be difficult and time-consuming, not scalable due to the test budget of CI environments. In this context, and to properly deal with the TCPCI problem, this work presents an approach based on Multi-Armed Bandit (MAB) called COLEMAN (*Combinatorial VOLatiLE Multi-Armed BANDit*). The TCPCI problem falls into the category of volatile and combinatorial MAB, because multiple arms (test cases) need to be selected, and they are added or removed over the cycles. We conducted an evaluation considering three time budgets and eleven systems. The results show the applicability of our approach and that COLEMAN outperforms the most similar approach from literature in terms of early fault detection and performance.

KEYWORDS

Test Case Prioritization, Continuous Integration, Multi-Armed Bandit

1 INTRODUCTION

Continuous Integration (CI) plays an important role in agile development, allowing reduced integration effort, lower number of uncorrected errors for long periods, and delivering a product version at any moment. CI environments are fundamental to CI in practice to support the frequent integration of changes and make the software evolution more rapid and cost-effective. Studies in the literature show that the adoption of such environments in the industry is growing, as well as the number of daily commits and the amount of automated tests as a consequence [38].

Companies like Google [27], Facebook, and Microsoft have adopted CI, as well as open-source projects [17] using available CI frameworks (i.e., Travis CI and Jenkins). A study shows that every day at Google, an amount of 800K builds, and 150 Million test runs are performed on more than 13K code projects [27]. This amount of builds and tests can require non-trivial amounts of time and resources [17]. Within an integration cycle (many times called build), the regression testing activity takes a significant amount of time. A test set, many times, includes thousands of test cases that take several hours or days to execute [15]. Even though massive parallelism was reported, Google developers must wait 45 minutes to 9 hours to receive testing results [27].

In this scenario, to re-execute all test cases is unfeasible. Then it is fundamental to perform Regression Testing (RT) activities in a very cost-effective way. This ensures that recent changes have not negatively impacted functionality previously tested, and considering CI goals, provides rapid test feedback on software failures. Besides, in a company, multiple projects may share the same CI workflow and regression testing usually runs a time restricted to a specific duration, the test budget. This makes difficult the use of traditional RT techniques for test case minimization and selection. They usually rely on code analysis and instrumentation, what can be time-consuming and produce results that quickly become inaccurate due to the frequent changes [12]. Test Case Prioritization (TCP) techniques are most suitable in the presence of time constraints. TCP techniques allow the most crucial test cases are executed first, by reordering a test suite according to specific goals, such as early fault detection. TCP techniques have advantages with respect to other techniques because they consider all test suite, consequently, decreasing the risk of reducing code coverage by discarding some test case [9].

Ideally, the test set should be executed to maximize early fault detection. However, fault-detection information is unknown until the testing is finished [37]. TCP techniques based on failure history can be used to overcome such a difficulty [15]. But a problem remains, related to the speed up of existing TCP approaches. Due to the high frequency of changes in CI environments and the high RT cost, approaches that require exhaustive analysis are costly and inefficient [12] because the time available to run the prioritized test suite can be reduced if prioritization takes too long [16].

To deal with the TCP problem in CI environments (TCPCI), some approaches have recently appeared in the literature addressing such difficulties: early fault detection and time constraints to run the test suite [6, 8, 15, 24, 25, 26, 33]. But few pieces of work consider CI particularities [29]. For instance, they do not consider the dynamic environment of the software life-cycle, in which changes in the

code are frequent and test cases can be added or removed (discontinued) over the CI cycles, particularly called test case volatility. Most approaches do not properly deal with an important dilemma called Exploration versus Exploitation (EvE). To solve such a dilemma an approach needs to balance: i) the diversity of test suite, and ii) the quantity of new test cases and test cases that are error-prone or that comprise high fault-detection capabilities. This problem is related with the test budget, because if only error-prone test cases are considered without diversity, some test cases can never be executed. To deal with this, existing approaches use, besides the failure-history, other measures that rely on code instrumentation or require additional information, such as to calculate code or feature coverage. This can be time-consuming, and to maintain the information updated can be difficult.

The approach called RETECS (Reinforced Test Case Selection) [33] deals with the EvE dilemma and considers the test budget by using historical test data and Reinforcement Learning (RL) [34]. RETECS reached the best performance using an Artificial Neural Network (ANN), which is capable of handling a large amount of decisions/states. On the other hand, determining why an ANN makes a particular decision is a hard task (black box) [4].

Considering the aforementioned limitations of existing work, in this paper, we introduce an approach for the TCPCI problem based on Multi-Armed Bandit (MAB) [2, 31]. The MAB problems are a class of sequential decision problems that are intensively studied for solving the EvE dilemma [20]. MAB has many similarities to RL and is considered to be a “lite” form (one-state) of RL. But MAB presents some advantages. MAB does not require context information and its actions only affect the reward, that is, the actions do not change the state of the environment. In contrast, RL actions change the state. Due to this, RL needs to handle the state space, as well as to rely on function approximation to evaluate the value of being in a particular state and taking a specific action.

The TCPCI problem falls into the category of volatile and combinatorial MAB, because multiple arms (test cases) need to be selected, and they are added or removed over the cycles. Then, we named our approach COLEMAN (*Combinatorial Volatile Multi-Armed Bandit*). We implemented our approach with five MAB policies and evaluated it in eleven large-scale real-world software systems, considering three time budgets. After that, we compared the best policy found for COLEMAN against RETECS [33]. Quality indicators, such as Normalized Average Percentage of Faults Detected (NAPFD), Average Percentage of Faults Detected with cost consideration (APFDC), as well as statistical tests and effect size, are used. Evaluation results show the applicability of our approach, which takes in all cases less than one second to execute, even for the systems with the greatest number of builds and test cases. COLEMAN outperforms RETECS with statistical difference in the majority of the cases, considering all budgets and indicators used. In summary, this work has the following main contributions:

(1) The proposal of COLEMAN, a MAB-based approach for the TCPCI problem. Such an approach has some advantages in comparison with related work:

- It learns how to incorporate the feedback from the application of the test cases thus incorporating diversity in the test suite prioritization;

- It uses a policy to deal with the EvE dilemma, thus mitigating the problem of beginning without knowledge (learning) and adapting to changes in the execution environment, for instance, the fact that some test cases are added (new test cases) and removed (obsolete test cases) from one cycle to another (volatility of test cases);
- It is model-free. The technique is independent of the development environment and programming language, and does not require any analysis in the code level;
- It is more lightweight, that is, needs only the historical failure data to execute, and has higher performance.

(2) As far as we know, COLEMAN is the first MAB-based approach that considers standard MAB with the characteristics of combinatorial and volatile MAB, even considering other fields beyond software testing.

(3) A public repository with the data used in this work, which allows replication and can be used in future research available in the Open Science Framework[28].

Paper structure. Section 2 provides background information about MAB and CI environments. Section 3 reviews related work on TCPCI. Our proposed approach, COLEMAN, is detailed in Section 4. Section 5 describes how the experiments were conducted: research questions, systems under test, quality indicators and used parameters. Results are presented and discussed in Section 6. Section 7 concludes the paper and presents the directions for future work.

2 BACKGROUND

In this section, we review background on Combinatorial and Volatile Multi-Armed Bandit and CI environments.

2.1 Multi-Armed Bandit

MAB (*Multi-Armed Bandit*) problems [31] are sequential decision problems related to the EvE (Exploitation versus Exploration) dilemma [20]. This means that, for such problems, solutions with the best performance (exploitation) are desired, but it is also important to ensure diversity, that is, dissimilar solutions (exploration).

The MAB problem is related to the scenario in which a player plays on a set K of slot machines (or arms/actions) that even identical produce different gains. After a player pulls one of the arms $i \forall i \in K$ in a turn t , a reward ($\hat{q}_{i,t}$) is received drawn from some unknown distribution, thus aiming to maximize the sum of the rewards.

A policy is a strategy that chooses, at each time t , the next arm to pull based on previously observed rewards and decisions. The MAB problem is to determine the policy that maximizes the expected cumulative reward over the EvE dilemma. A review of the main MAB policies proposed in the literature is presented in [20]. Among them, we can mention the ϵ -greedy policy, a policy widely used due to its simplicity. At each time, such a policy evaluates the arms and defines an empirical quality estimate $\hat{q}_{i,t}$, based on previous executions. The $\hat{q}_{i,t}$ value of an arm i in the time t is based on the sum of its previous rewards divided by the number of times that i has been pulled. After, the policy selects, with a probability $1 - \epsilon$, the arm with the highest $\hat{q}_{i,t}$ value (exploitation), or randomly selects

an arm (exploration) with a probability ϵ . The parameter ϵ is the key to balance the EvE dilemma in the ϵ -greedy policy.

The MAB-policy called Upper Confidence Bound (UCB) provides a smarter way to deal with the EvE dilemma and ensures asymptotic optimality in terms of the total cumulative reward [2]. Most of the recent policies are UCB-based. In such policies the i th arm has an empirical quality estimate $\hat{q}_{i,t}$ (the average of the rewards obtained up to the given time instant) and a confidence interval that depends on the number of times, n_i , the arm has been applied before. At each time point t , the selection of the best arm is performed based on the arm with the best upper bound of the confidence interval. In addition to this, Fialho [13] introduced a scaling factor C (Equation 1), encompassing the situation when the rewards are usually among some real-value interval and the EvE balance may “break”.

$$\text{Select } a_t = \underset{i \in K}{\operatorname{argmax}} \left(\hat{q}_{i,t} + C \times \sqrt{\frac{2 \times \ln \sum_{j=1}^K n_{j,t}}{n_{i,t}}} \right) \quad (1)$$

where the exploitative first term favors the arms with best empirical rewards, while the exploratory second term favors the infrequently tried arms. If exploration is preferable, C must be increased. Otherwise, if exploitation is the focus, C must be decreased. In this work, the acronym UCB is used to denominate the adaptation proposed by Fialho.

Another UCB-based MAB policy is the Fitness-Rate-Rank (FRRMAB), a state policy that has presented good results in the Adaptive Operator Selection context [21]. This policy consists of two procedures: credit assignment and operator (arm/action) selection. The first procedure, *Credit Assignment*, refers to a reward procedure that takes into account the impact observed in the most recent applications. In the credit assignment, FRRMAB policy changed the UCB quality estimator (Equation 1) by a rank-based method that uses the Fitness Improvement Rate (FIR) method. The FIR value is stored in a given Sliding Window (SW) organized as a first-in, first-out queue that is used to evaluate the W recent applications. The FRRMAB final value is corrected by a decaying factor (DF). The second procedure randomly selects an arm until all arms are selected, uses the FRRMAB policy to evaluate each arm, and selects the best one.

In many real-world scenarios, a policy needs to select multiple arms in each time. This is the case in our scenario, TCPCI problem, where we need to prioritize a test set, and we can consider a test case as an arm. This kind of MAB problem is categorized as *combinatorial bandit*, in which a set of arms are chosen in each time t rather than one individual arm, that is, we are required to pull a fixed number m of arms from a set of arms K , such that $1 \leq m \leq |K|$ [1].

In addition to this and considering the inherently dynamic nature of our problem, the arms available in each time may change dynamically over time. In this sense, this work is based on a MAB variant known as *Volatile-multi-Arm bandit* (VMAB) [5]. VMAB considers that the arms can “appear” or “disappear” unexpectedly in each time. In VMAB each arm a_i is associated with a *lifespan* given by an interval of time (t_{start_i}, t_{end_i}) , during which this arm is available. The arm’s *lifespan*s are unknown in advance.

In this paper, we propose a novel approach that combines combinatorial MAB and VMAB. It allows the use of multiple plays in an adaptive (volatile) scenario.

2.2 Continuous Integration environments

We observe an increasing use of Continuous Integration (CI) environments in the industry [10]. CI environments automate the process of building and testing software, allowing engineers to merge code that is under development or maintenance with the mainline code base at frequent time intervals [10]. In CI development, teams work continuously integrating code and make smaller code commits every day, usually monitored by a CI server. When a change occurs, the CI server clones this code, builds it and runs the testing processes. When the entire process is finished, a report (feedback) is generated by the CI server, and the developers are informed. Figure 1 illustrates the process.

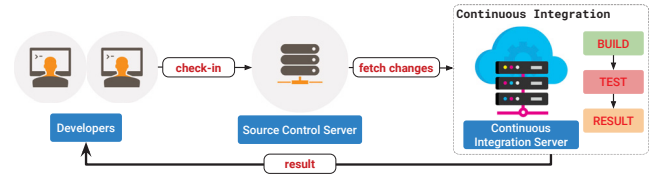


Figure 1: Overview of a Continuous Integration environment.

CI automated support is very important to CI in practice. Zhao et al. [38] present some results about the impact of adopting the framework Travis CI on development practices in a collection of GitHub projects. They observed an increase in the number of daily commits (frequency of 78 commits every day) and, after some initial adjustments, an increase in the amount of automated tests.

In this scenario, RT activities are fundamental and TCP techniques have been adopted to deal with the test budget, limited resources and constraints of CI environments. In the next section, we review related work on TCPCI.

3 RELATED WORK

There are many works on TCP in the literature, which have been subject of surveys and mappings [7, 18, 37]. However, only few pieces of work address CI environments. Some approaches have the goal of reducing the amount of resources utilized in a CI environment. Liang et al. [22] propose an approach to prioritize commits based on the test suite failure and execution history. Other approach that considers multiple test requests was proposed by Zhu et al. [39]. Such an approach uses co-failure distributions of tests, that is, tests that co-fail in previous executions in different sets. Elbaum et al. [12] introduce new algorithms for selection and prioritization of test suites, to be used respectively in a pre-submit and pos-submit testing phases. In the first phase, which occurs prior to commit, developers specify modules to be tested and the selection algorithm selects test sets based on failure and execution windows. This allows, respectively, selection based on the fault detection history and of test cases not recently executed. In the post-testing, after commit, the TCP algorithm prioritizes test suites considering

both windows and a time window. In this way, the algorithm deals with test suite concurrency execution in the pos-submit testing.

Differently from the mentioned works, our work assumes a sequential execution order of test cases and prioritizes test cases in the suite to be executed after the commit, usually considering a test budget. Approaches with such a goal are the most related to ours, and are described next.

Marijan et al. [25] introduce ROCKET, an approach that, given a test budget, sets a weight for each test case based on the distance of the failure status from its current execution and its execution time. We observe a limitation regarding the execution time. Test cases with an execution time greater than a limit are penalized, and it is possible they are never executed. To set the weight, neither the prioritization feedback is considered nor the total history of failures. An extension is proposed by the authors in [24] to consider other perspectives regarding fault detection, business, performance and technical aspects. Such an extension needs additional information related to coverage and features.

An approach and a tool called TITAN is proposed in [26] for Highly Configurable Software (HCS). It implements test prioritization and minimization techniques, and provides test traceability and visualization. Again, additional information, such as feature coverage, is necessary. As first a minimization step is conducted the prioritized set may not contain all available test cases.

Cho et al. [8] proposed an approach, named AFSAC, composed by two stages. First, weights for the test cases are determined by using statistical analysis over the failure history. Then, the test cases are reordered using the correlation data of test cases acquired by previous test results.

Haghighatkhan et al. [15] present empirical results that show the use of historical failure knowledge is a strong predictor for TCPPI problem. It is effective to catch regression faults earlier without requiring a large amount of historical data. In addition to this, the effectiveness can be improved by using such a knowledge with a diversity measure, calculated by comparing the text of test cases.

The above-mentioned approaches do not consider the volatility of the test cases, that is, they do not consider a test case may added and/or removed over the cycles. Most of them do not consider the feedback from the prioritization conducted previously. Some works have addressed this limitation with the use of machine learning. The approach of Busjaeger and Xie [6] uses (SVM^{map}) to create a model based on five attributes: test coverage of modified code, textual similarity between tests and changes, recent test-failure or fault history, and test age. Such a model is used to predict the fault-proneness of the test cases and prioritize them. However, to obtain the used attributes, additional information and code instrumentation are necessary. Other approach is RETECS, introduced by Spieker et al. [33] to prioritize and select test cases based on Reinforcement Learning (RL). RETECS considers as input the test case duration, historical failure data, and previous last execution. The authors compared different RL variants and the Artificial Neural Network (ANN) variant presented the best results.

We can summarize some drawbacks identified in related work. Some of them have different goals from ours, for instance to reduce server resources considering concurrent test set executions. The most related approaches do not properly deal with the EvE problem.

This problem regards to the fact that as only a sub-set of the prioritized test cases can be executed regarding its order, some test cases can never be executed given the test budget. To deal with this, most approaches use, besides the failure-history, other measures that rely on code instrumentation or require additional information, such as to calculate code or feature coverage. This can be time-consuming and to maintain the information updated can be difficult.

Existing works do not take into consideration the volatility of test cases and feedback from last prioritizations. Differently, our approach considers the test cases volatility and learns with the past prioritizations (*online learning*). It properly deals with the EvE dilemma without requiring source code analysis or any initial concept (model) about the system. In this sense, RETECS [33] is the most similar to ours. But RETECS has the best performance using an ANN [33], which usually requires a large amount of data [40]. Moreover, determining why an ANN makes a particular decision is a hard task. This makes difficult to trust in their reliability for real-world problems [4]. Our approach uses MAB, which allows test cases rewards in a sliding window, and works with less input information. Context information is not necessary. In this way, we aim to properly deal with the EvE dilemma with an approach more lightweight and with higher performance.

4 PROPOSED APPROACH

In this section we describe our test case prioritization approach, named COLEMAN (*Combinatorial VOLatiLE Multi-Armed BANDit*). Given the dynamic nature of our problem, our MAB approach combines two MAB variants: i) combinatorial MAB, because we have a set of arms (test cases), and ii) volatile, because at a given time t such set varies, test cases can be added or removed over the software life-cycle. In addition to this, the approach works with a budget (constraint) to execute the test cases prioritized. To ensure diversity of test cases to be executed, it uses MAB policies, allowing better exploitation and exploration (EvE dilemma).

Figure 2 illustrates how our approach works in a CI environment. After a successful build, in the test phase, the approach receives as input a set of test cases T_t (arms) available for the current commit (cycle/time) t and, based on the choice order given by a MAB policy, generates the prioritized test case set T'_t . In each time t , only one test suite (test case set) is prioritized. Then the system is tested using T'_t and feedback from this test set is collected, containing information, such as the test cases executed in a time limit, the number of failures, the test case failure rank, and so on. This feedback is used by the reward function in the credit assignment procedure to set individual rewards for each test case. In the T'_t evaluation, a fitness value of T'_t is obtained by a quality indicator.¹ This value can be used by testers along with the commits to evaluate the prioritization quality.

Then, the credit assignment procedure calculates the rewards for each arm, test case $t'_c \in T'_t$, which are stored in a historical database to be used in the next commits by the policies. In the end, results are reported back to the CI server. The CI server sets the result from the cycle and notifies the parties interested in the cycle.

Next, we detail the main elements of COLEMAN: MAB policies (Section 4.1) and credit assignment (Section 4.2).

¹In this work we use NAPFD (see Section 5.1).

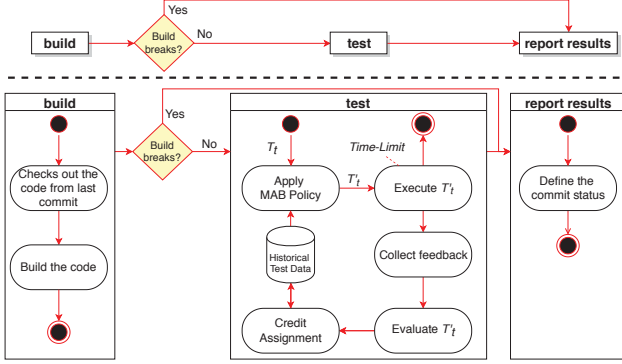


Figure 2: Overview of the proposed approach and how it is integrated in the test phase of a CI environment.

4.1 MAB Policies

Considering the traditional behavior of a MAB policy, the best arm (test case) t_c from a set T_t at each time t is chosen and applied. However, we are working with TCP problem where a prioritized test set is used. To this end, we can adapt a policy to choose the best arm (test case) t_c from T_t to compose T'_t , then remove t_c from T_t , and continue this process until no more test cases are available in T_t . An arm t_c is chosen only once and the order of choice defines the prioritization execution. But this process of choice is costly, once that a new evaluation for each test case in T_t is necessary to choose the next best test case to compose T'_t .

To reduce the selection cost, we adapted the policies to evaluate all the test cases (arms), and order them, putting the best test case in the top, followed by the second best one, and so on. When more than one test case has the same *performance*, the order among them is defined randomly. This simple adaption allows a fast prioritization whilst considers the characteristics of the policy chosen. We also adapt the chosen policy to consider only the test cases available in time t and ignore the other ones from the previous time. This modification allows us to consider the dynamic environment (volatility) of the test cases.

As mentioned in Section 2.1, there are many MAB policies. We chose the policies that better work with the EvE dilemma: ϵ -greedy, UCB, and FRRMAB. Besides these policies, we also assessed the following policies as baselines: i) Random: is a strategy (not a real strategy per se) where the player will only do exploration; and ii) Greedy: only takes the best apparent arm, and it is a special case of ϵ -greedy where $\epsilon = 0$, i.e., it always does exploitation.

It is important to highlight that FRRMAB policy works with a sliding window. The reward value (FIR for FRRMAB) is obtained through a reward function (Section 4.2.1), then the last W rewards are used by the policy. In this way, for each test case, FRRMAB policy considers the history of rewards whilst the other ones use cumulative rewards.

4.2 Credit Assignment

This procedure reflects the goal of the prioritization and teaches the MAB-based policy about the test cases considering historical test data. In this procedure, for $t'_c \forall t'_c \in T'_t$, two values are assigned:

the number of times $n_{t'_c}$ has been applied before and the reward value ($\hat{q}_{t'_c,t}$). These values are used by the policies to generate the prioritized test set. At the beginning (where $t = 1$) and for each new test case, the values for $n_{t'_c}$ and $\hat{q}_{t'_c,t}$ are assigned with zero. After that, the values are assigned as follows.

The number of times $n_{t'_c}$ that t'_c has been applied before is considered to explore new test cases (few used). Traditionally, a MAB policy selects an arm and increments the number of times that this arm was chosen. In our case, we use a combinatorial MAB, and this kind of MAB selects a set of arms (test cases). To counterbalance the order of choice, a weight is given for each $t'_c \in T'_t$ according to its order in T'_t . The weights are evenly spaced values within an interval (0.0, 1.0) with a step size of $\frac{1}{|T'_t|}$ in descending order. In

this way, the highest weight is given to the first test case in T'_t and the lowest to the last one. Then, $n_{t'_c}$ is incremented with the weight defined to t'_c .

The reward value is obtained by a reward function (Section 4.2.1). This value is used to exploit the best test cases. As described in Section 4.1, the reward value is stored in a sliding window when FRRMAB policy is used whilst ϵ -greedy, greedy, and UCB policies use a cumulative reward strategy. If a new test case appears, a zero is set for the reward value, once that we do not have a *test case history*. On the other hand, if a test case is removed in the current cycle (commit), we remove its history.

4.2.1 Reward Functions. In this work, we adopt and adapted two reward functions from related work [33]. The first reward function *RNFail* (Reward Based on Failures) is based on the number of failures associated with a test case $t'_c \in T'_t$ and uses the function *fails* defined in Table 1.

Table 1: Functions used by the Reward Functions.

Definition	Description
$fails(t'_c)$	In our context, a test case t'_c can be composed by many parts (or test methods), each one of this part can be associated with a failure. In this way, a failing test case t'_c can be associated with one or more failures. Function <i>fails</i> (t'_c) returns the number n_f of failures associated with t'_c .
$fails(t'_c)$	The function <i>fails</i> (t'_c) returns 1 if <i>fails</i> (t'_c) ≥ 1 , and 0 otherwise.
$rank(t'_c)$	The function <i>rank</i> (t'_c) returns the position of t'_c in a prioritized set T'_t .
$prec(t'_c, t'_{c2})$	The function <i>prec</i> (t'_c, t'_{c2}) returns 1 if <i>rank</i> (t'_c) < <i>rank</i> (t'_{c2}).

$$RNFail(t'_c) = \begin{cases} 1 & \text{if } fails(t'_c) \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The second reward function *TimeRank* (Time-Ranked Reward) is based on the rank of $t'_c \in T'_t$ (Equation 3). The idea is to evaluate whether failing test cases, with a greater number of failures, are ranked in the first positions in T'_t . To this end, a test case t'_c that does not fail and precedes failing test cases is penalized by their early scheduling.

$$TimeRank(t'_c) = \frac{|T'^{fail}| - [\neg(fails(t'_c)) \times \sum_{i=1}^{|T'^{fail}|} prec(t'_c, t'_{ci})]}{|T'^{fail}|} \quad (3)$$

where T'^{fail} is composed by the test cases of T'_t that failed. A non failed test case receives a reward given by the accumulated number of test cases which failed until its position in the prioritization rank,

that is, it receives a reward decreased by the number of failing test cases ranked after it in the rank.

5 EVALUATION DESCRIPTION

The main hypothesis of this work is that MAB can be used to address the TCPFI problem in a very cost-effective way. Then, our experiment evaluates the COLEMAN applicability and performance in CI environments. We also perform a comparison with related work. The experiment is guided by the following research questions:

- RQ1:** *What is the best configuration for COLEMAN?* This question aims to identify the best MAB policy and reward function to be used with COLEMAN.
- RQ2:** *Is COLEMAN applicable in the CI development context?* This question is specially important for software testers that want to use COLEMAN in practice. It investigates whether the time spent in the prioritization is acceptable considering CI Cycles (commits).
- RQ3:** *Can COLEMAN outperform RETECS?* This question compares our approach, with RETECS, the RL approach, which is the most similar to COLEMAN.

To answer RQ1, we compare five MAB policies: Random, ϵ -Greedy, Greedy, UCB, and FRRMAB, (Section 4.1) and both reward functions: *RNFail* and *TimeRank* (Section 4.2.1). We use indicators commonly applied in TCP (Section 5.1). We evaluate three time constraints (budgets) considering: 10%, 50%, and 80% of the execution time of the overall test set available in each commit. The best policy identified in RQ1 is used to answer the remaining questions.

To answer RQ2, we compare the prioritization time spent by COLEMAN and the time between commits, as well as the percentage of reduced time to test execution.

To answer RQ3, we execute the implementation of RETECS available in the literature² by using ANN, which obtained the best results in comparison with a Tableau representation [33]. Our approach considers only a minimal information to prioritize the test cases, historical failure data, whilst RETECS needs additional information concerning each test case: duration, the time it was last executed, and results from its previous execution (passed or failed).

In this experiment, if a test case is removed in a commit, it is then removed along with its history. The results are obtained from 30 independent executions for each system, reward function, and time budget. All the experiments are performed on an Intel® Xeon® E5-2640 v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS.

5.1 Quality Indicators

To evaluate the performance of the approaches concerning failure detection effectiveness of a test suite T , we use NAPFD [30] (Equation 4) and APFDc [11] (Equation 5). Both metrics are extensions of the Average Percentage of Faults Detected (APFD) [32]. APFD indicates how quickly a set of prioritized test cases (T') can detect the faults present in the application being tested, and its value is calculated from the weighted average of the percentage of detected faults. APFD values range from zero to one. Higher values indicate

that the faults are detected faster using fewer test cases. To calculate NAPFD, we consider the ratio between detected and detectable faults within T . This metric is adequate for prioritization of test cases when not all of them are executed, and some faults can be undetected.

$$NAPFD(T'_t) = p - \frac{\sum_1^n rank(T'_t)}{m \times n} \frac{p}{2n} \quad (4)$$

where m is the number of faults detected by all test cases; $rank(T'_t)$ is the position of T'_t in T' , if T'_t did not reveal a fault we set $T'_t = 0$; n is the number of tests cases in T' ; and p is the number of faults detected by T' divided by m . The last part of the equation represents the full area under the curve when the percentage of faults found is plotted on the y-axis and the percentage of the run test cases is on the x-axis. NAPFD is equal to APFD metric if all faults are detected.

The APFDc metric assumes that the test cases do not have the same cost, that is, some may be more costly to execute than others, maybe due to the fault severity or running time. The test cases are usually prioritized until a maximum cost is reached, which is feasible to execute. Furthermore, if both fault severity and test case costs are identical, APFDc can be used to compute the APFD value. In this work, we consider that all faults have same severity.

$$APFDc(T'_t) = \frac{\sum_{i=1}^m (\sum_{j=TF_i}^n c_j - 0.5c_{TF_i})}{\sum_{j=1}^n c_j \times m} \quad (5)$$

where c_j is the cost of a test case T_i , and TF_i is the first test case from T' that reveals fault i .

To evaluate the test suite efficiency concerning how fast it is to detect a fault, we use the Rank of the Failing Test Cases (RFTC). In this rank, lower values represent a faster failure detection. In this sense, we extract the order of the first test case that fails from the prioritized test suite. This metric is useful when we need fast feedback from test cases and there is only one fault that the test cases are seeking.

Furthermore, we define a metric (Equation 6) named Normalized Time Reduction (NTR), in order to observe the difference between time spent until the first test case fails r_t and the total time spent to execute all tests \hat{r}_t . In this metric, only the commits that failed, $CIfail$, are considered. In this way, we can evaluate the capability of an algorithm to reduce the time spent in a CI cycle.

$$NTR(\mathcal{A}) = \frac{\sum_{t=1}^{CIfail} (\hat{r}_t - r_t)}{\sum_{t=1}^{CIfail} (\hat{r}_t)} \quad (6)$$

The last indicator used is Prioritization Time. Such measure computes the time spent (in seconds) by an algorithm to perform the prioritization. This value helps to observe whether an approach spends much time, what can make it impracticable for real scenarios.

We apply Kruskal-Wallis [19], Mann-Whitney [23], and Friedman [14] statistical tests with a confidence level of 95%. We use Kruskal-Wallis to evaluate the performance of the approaches in each system over 30 independent runs. We use Mann-Whitney to evaluate a pair of performances in the same system or for post-hoc analysis. We use Friedman to evaluate the approach behavior across different systems. To this end, each system becomes a dependent variable, in which we apply multiple approaches.

²<https://bitbucket.org/helges/RETECS>

Additionally, to calculate the effect size magnitude of the difference between two groups, we use the Vargha and Delaney’s \hat{A}_{12} [36] metric. This measure ranges from 0 to 1 and defines the probability of a value, taken randomly from the first sample, is higher than a value taken randomly from the second sample. A *Negligible* magnitude ($\hat{A}_{12} < 0.56$) represents a very small difference among the values and usually does not yield statistical difference. The *Small* ($0.56 \leq \hat{A}_{12} < 0.64$) and *Medium* ($0.64 \leq \hat{A}_{12} < 0.71$) magnitudes represent small and medium differences among the values, and may or not yield statistical differences. Finally, a *Large* magnitude ($0.71 \leq \hat{A}_{12}$) represents a significantly large difference that usually can be seen in the numbers without much effort.

5.2 Systems Under Test

We select non-toy, non-fork, and active GitHub (GH) projects considering *watchers* and *stars* on GH, as well as some systems already used in the literature [15, 33]. Most of them using Travis CI [35] and Maven.³ To collect the Travis CI build history, we adapt and use TravisTorrent⁴ [3].

The systems under test (SUTs) are detailed in Table 2. The second column shows the period of build logs analyzed. The third column presents the total of builds identified, and in parenthesis the number of builds included in the analysis. We discard build logs with some problem, identified by Travis CI, such as the ones related to non-valid build logs and test cases that did not execute. The fourth column shows the total of failures found, and in parenthesis the number of builds in which at least one test failed. The fifth column shows the number of different (unique) test cases identified from build logs, and in parenthesis the range of test cases executed in the builds. The last columns present, for each system, the mean (\pm standard deviation) duration in minutes of the CI Cycles and the interval between them.

The systems IOF/ROL, Paint Control, and GSDTSR are selected for comparison because they are the same systems used in related work to evaluate RETECS [33]. However, in the related work the datasets are analyzed considering that a CI Cycle includes all the test cases executed per day. In our study, we consider the CI Cycle as a commit. In this way, we change the datasets representations to consider each date in the last run information is a commit. These systems have different characteristics (number of faults, test cases, and commits), and they can ensure the evaluation of the approaches in relation to the generalization capacity.

5.3 Parameters Setting

RETECS is available online and it is evaluated using ANN with the default values defined in related work [33], for Hidden Nodes, Replay Memory, and Replay Batch Size, with, respectively, 12, 10000, and 1000. In this way, a tuning phase was necessary only to choose the parameters values for the MAB policies UCB, FRRMAB, and ϵ -Greedy. They are the scaling factor C to control the trade-off among EvE for UCB and FRRMAB policies, the sliding window size W and decayed factor DF for FRRMAB, and the probability ϵ for

ϵ -Greedy policy. The possible values used in this phase for C are 0.3, 0.5, 1.0, and 2.0, for W were 10, 50, 100, 150, 200, 250, and 300, and for ϵ 0.1, 0.2, 0.5, 0.7, and 0.9. After some empirical studies, the decayed factor is no longer used and a default value equals to 1 is defined.

We conduct an empirical evaluation with 10 independent runs for each MAB policy with different parameters. For this evaluation, we consider the systems DeepLearning4j and Fastjson. These systems are chosen because they have the best trade-off between the mean number of failures by CI Cycles and the number of CI Cycles. Additionally, we define a test budget for a CI Cycle with a fixed percentage of 50% of the required time as adopted in [33]. The best parameters found for the MAB policies are: ϵ -Greedy and UCB with 0.5 (in the *TimeRank* function) and 0.3 (*RNFail*) for parameter C ; and FRRMAB with 0.3 for C and 100 for W .

5.4 Threats to Validity

We identify the following points that can be threats to the validity of the results. The first threat is the parameter configuration of the algorithms. Other parameters can lead to different results. To mitigate this threat, we empirically evaluate different ranges of parameters for the MAB policies and for RETECS we adopt the configuration of related work, since we are using the same systems.

The dataset representation used is a threat. We change information about the CI Cycles for systems IOF/ROL, Paint Control, and GSDTSR. This change can impact the results, mainly because we do not know with precision what information is from each commit and whether RETECS is deeply dependent on this kind of representation. For this reason, we group the data by the last run date.

6 RESULTS AND ANALYSES

In this section, the experimental results are presented and analyzed aiming to answer the posed questions. Supplementary material with datasets, results, and additional analysis can be found in our public repository [28].

6.1 RQ1: COLEMAN Configuration

As mentioned before, to answer RQ1, we compare the five MAB policies with both reward functions taking account three budgets. We use the indicators: NAPFD, APFDc, and RFTC. The results and a complete analysis are available in supplementary material [28]. Due to lack of space, we detail only NAPFD and APFDc results regarding the budget of 50% (Table 3). According to Spieker et al. [33], a time budget of 50% presents a constraint that allows better comparison whilst keeps the difficulty inherent from the problem.

The average is computed using results from 30 independent executions found by each policy in each SUT. Values highlighted in bold are the best, and values that are statistically equivalent to the best ones have their corresponding cells painted in light gray. Furthermore, we use different symbols to indicate the effect size magnitude concerning the best values. For each comparison in each SUT, we use the Kruskal-Wallis test. When detected statistical difference, we apply a post-hoc analysis using the Mann-Whitney test with Bonferroni p -value adjustment method to find the statistical difference among the policies. In order to evaluate the performance of the policy concerning all systems, we compute the average across

³Maven is a build automation tool used primarily for Java projects. We choose projects which use Maven as a testing framework because it provides detailed output traces (more verbose).

⁴<https://github.com/jacksonpradolima/travistorrent-tools>

Table 2: Test Case Set Information.

Name	Period	Builds	Failures	Test Cases	Duration (min)	Interval (min)
Druid	2016/04/24-2016/11/08	286 (168)	270 (71)	2391 (1778-1910)	4.27 ± 10.66	384.76 ± 468.86
Fastjson	2016/04/15-2018/12/04	2710 (2371)	940 (323)	2416 (900-2102)	1.97 ± 0.89	233.22 ± 401.26
Deeplearning4j	2014/02/22-2016/01/01	3410 (483)	777 (323)	117 (1-52)	12.33 ± 14.91	306.05 ± 442.55
DSpace	2013/10/16-2019/01/08	6309 (5673)	13413 (387)	211 (16-136)	11.78 ± 7.03	291.29 ± 411.19
Guava	2014/11/06-2018/12/02	2011 (1689)	7659 (112)	568 (308-512)	62.53 ± 80.31	435.55 ± 464.52
OkHttp	2013/03/26-2018/05/30	9919 (6215)	9586 (1408)	289 (2-75)	7.64 ± 5.64	220.17 ± 405.93
Retrofit	2013/02/17-2018/11/26	3719 (2711)	611 (125)	206 (5-75)	2.40 ± 1.60	270.86 ± 449.41
ZXing	2014/01/17-2017/04/16	961 (605)	68 (11)	124 (81-123)	13.14 ± 12.37	411.10 ± 465.53
IOF/ROL	2015/02/13-2016/10/25	2392 (2392)	9289 (1627)	1941 (1-707)	1537.27 ± 2018.73	1324.26 ± 291.75
Paint Control	2016/01/12-2016/12/20	20711 (20711)	4956 (1980)	1980 (1-74)	424.46 ± 275.90	1417.86 ± 144.97
GSDTSR	2016/01/02-2016/02/01	259388 (259388)	3208 (2924)	5555 (1-390)	974.25 ± 4850.66	1439.91 ± 2.58

Table 3: NAPFD and APFDc values (mean and standard deviation) for MAB policies using time budget of 50%.

This table reports the NAPFD and APFDc results (averages ± standard deviation) obtained from 30 independent runs with time budget of 50% and organized by each Reward Function under study (see Section 4.2.1). Values highlighted in bold with a “★” symbol denotes the best algorithm for a Reward Function in a SUT and, in gray, results that are statistically equal to the best one. A “▼” indicates that the effect size was negligible in relation to the best, while “▽” denotes a small magnitude, “△” a medium magnitude, and “▲” a large magnitude. The effect size was performed during the post-hoc tests, that is, when there is a statistical difference.

SUT	NAPFD					APFDc				
	FRRMAB	UCB	ε-Greedy	Greedy	Random	FRRMAB	UCB	ε-Greedy	Greedy	Random
RNFail - Reward based on Failures										
Druid	0.9333 ± 0.013 △	0.9422 ± 0.007 ★	0.8472 ± 0.110 ▲	0.8965 ± 0.072 ▽	0.7464 ± 0.014 ▲	0.9486 ± 0.016 ★	0.9486 ± 0.007 ▼	0.8477 ± 0.110 ▲	0.8979 ± 0.073 △	0.7506 ± 0.014 ▲
Fastjson	0.9174 ± 0.021 ▲	0.9597 ± 0.001 ★	0.9501 ± 0.005 ▲	0.9507 ± 0.005 ▲	0.9176 ± 0.003 ▲	0.9186 ± 0.021 ▲	0.9595 ± 0.001 ★	0.9491 ± 0.006 ▲	0.9498 ± 0.005 ▲	0.9193 ± 0.002 ▲
Deeplearning4j	0.7890 ± 0.001 ▲	0.7911 ± 0.002 ▲	0.8066 ± 0.003 ▲	0.8084 ± 0.002 ★	0.6381 ± 0.011 ▲	0.8106 ± 0.001 ★	0.7971 ± 0.002 ▲	0.7974 ± 0.004 ▲	0.7961 ± 0.003 ▲	0.6404 ± 0.016 ▲
DSpace	0.9724 ± 0.009 ★	0.9720 ± 0.001 ▼	0.9692 ± 0.002 ▼	0.9685 ± 0.002 ▼	0.9581 ± 0.001 ▲	0.9737 ± 0.009 ★	0.9730 ± 0.001 ▼	0.9713 ± 0.002 ▼	0.9704 ± 0.002 ▼	0.9587 ± 0.001 ▲
Guava	0.9653 ± 0.004 ▲	0.9750 ± 0.002 △	0.9768 ± 0.006 ★	0.9761 ± 0.010 ▼	0.9603 ± 0.002 ▲	0.9687 ± 0.003 ▲	0.9756 ± 0.002 ▽	0.9770 ± 0.006 ★	0.9758 ± 0.010 ▼	0.9611 ± 0.002 ▲
OkHttp	0.9192 ± 0.000 ★	0.9023 ± 0.001 ▲	0.9041 ± 0.005 ▲	0.8993 ± 0.006 ▲	0.8513 ± 0.002 ▲	0.9177 ± 0.000 ★	0.9039 ± 0.002 ▲	0.8994 ± 0.005 ▲	0.8950 ± 0.006 ▲	0.8549 ± 0.002 ▲
Retrofit	0.9853 ± 0.000 ★	0.9799 ± 0.001 ▲	0.9717 ± 0.001 ▲	0.9717 ± 0.001 ▲	0.9710 ± 0.002 ▲	0.9850 ± 0.000 ★	0.9794 ± 0.001 ▲	0.9722 ± 0.001 ▲	0.9722 ± 0.001 ▲	0.9712 ± 0.001 ▲
ZXing	0.9846 ± 0.000 ▲	0.9876 ± 0.000 ▲	0.9873 ± 0.002 ▲	0.9869 ± 0.002 ▲	0.9892 ± 0.002 ★	0.9862 ± 0.000 ▲	0.9877 ± 0.000 ▲	0.9876 ± 0.001 ▲	0.9873 ± 0.001 ▲	0.9893 ± 0.001 ★
IOF/ROL	0.5046 ± 0.002 ★	0.4791 ± 0.003 ▲	0.4792 ± 0.002 ▲	0.4790 ± 0.002 ▲	0.4786 ± 0.002 ▲	0.5081 ± 0.002 ★	0.4794 ± 0.003 ▲	0.4796 ± 0.002 ▲	0.4793 ± 0.002 ▲	0.4788 ± 0.002 ▲
Paint Control	0.9150 ± 0.000 ★	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9162 ± 0.000 ★	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲
GSDTSR	0.9893 ± 0.000 ★	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9894 ± 0.000 ★	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9892 ± 0.000 ▲
Average	0.8978	0.8993	0.8905	0.8946	0.8558	0.9021	0.9007	0.8895	0.8934	0.8571
TimeRank - Time-Ranked Reward										
Druid	0.9710 ± 0.008 ★	0.8422 ± 0.066 ▲	0.8767 ± 0.040 ▲	0.8503 ± 0.048 ▲	0.7489 ± 0.014 ▲	0.9787 ± 0.009 ★	0.8768 ± 0.078 ▲	0.8988 ± 0.042 ▲	0.8871 ± 0.056 ▲	0.7534 ± 0.014 ▲
Fastjson	0.9118 ± 0.028 ▲	0.9544 ± 0.002 ★	0.9455 ± 0.004 ▲	0.9240 ± 0.009 ▲	0.9181 ± 0.003 ▲	0.9140 ± 0.027 ▲	0.9565 ± 0.002 ★	0.9516 ± 0.004 ▲	0.9303 ± 0.009 ▲	0.9199 ± 0.003 ▲
Deeplearning4j	0.8200 ± 0.000 ★	0.7193 ± 0.002 ▲	0.7028 ± 0.004 ▲	0.7047 ± 0.002 ▲	0.6366 ± 0.007 ▲	0.8134 ± 0.001 ★	0.7879 ± 0.004 ▲	0.7774 ± 0.007 ▲	0.7805 ± 0.004 ▲	0.6405 ± 0.012 ▲
DSpace	0.9766 ± 0.008 ★	0.9659 ± 0.001 ▲	0.9606 ± 0.002 ▲	0.9602 ± 0.002 ▲	0.9582 ± 0.001 ▲	0.9767 ± 0.009 ★	0.9683 ± 0.001 △	0.9646 ± 0.002 ▲	0.9649 ± 0.002 ▲	0.9588 ± 0.001 ▲
Guava	0.9675 ± 0.007 ▲	0.9698 ± 0.002 ▲	0.9738 ± 0.002 ★	0.9734 ± 0.004 ▼	0.9608 ± 0.002 ▲	0.9672 ± 0.007 ▲	0.9740 ± 0.003 ▲	0.9786 ± 0.003 ▲	0.9824 ± 0.004 ★	0.9614 ± 0.002 ▲
OkHttp	0.9317 ± 0.000 ★	0.8753 ± 0.001 ▲	0.8725 ± 0.002 ▲	0.8677 ± 0.004 ▲	0.8514 ± 0.002 ▲	0.9246 ± 0.000 ★	0.8930 ± 0.001 ▲	0.8881 ± 0.002 ▲	0.8866 ± 0.004 ▲	0.8550 ± 0.002 ▲
Retrofit	0.9893 ± 0.000 ★	0.9789 ± 0.001 ▲	0.9715 ± 0.002 ▲	0.9689 ± 0.001 ▲	0.9707 ± 0.001 ▲	0.9885 ± 0.000 ★	0.9798 ± 0.001 ▲	0.9733 ± 0.002 ▲	0.9712 ± 0.001 ▲	0.9706 ± 0.001 ▲
ZXing	0.9857 ± 0.000 ▲	0.9879 ± 0.001 ▲	0.9882 ± 0.002 ▽	0.9861 ± 0.001 ▲	0.9891 ± 0.001 ★	0.9869 ± 0.000 ▲	0.9880 ± 0.001 ▲	0.9882 ± 0.002 △	0.9861 ± 0.001 ▲	0.9894 ± 0.001 ★
IOF/ROL	0.5189 ± 0.002 ★	0.4787 ± 0.002 ▲	0.4789 ± 0.002 ▲	0.4786 ± 0.002 ▲	0.4785 ± 0.002 ▲	0.5223 ± 0.002 ★	0.4789 ± 0.002 ▲	0.4792 ± 0.002 ▲	0.4789 ± 0.002 ▲	0.4786 ± 0.002 ▲
Paint Control	0.9150 ± 0.000 ★	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9162 ± 0.000 ★	0.9146 ± 0.000 ▲	0.9146 ± 0.000 ▲	0.9145 ± 0.000 ▲	0.9145 ± 0.000 ▲
GSDTSR	0.9894 ± 0.000 ★	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9894 ± 0.000 ★	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲	0.9891 ± 0.000 ▲
Average	0.9070	0.8796	0.8795	0.8743	0.8560	0.9071	0.8915	0.8912	0.8883	0.8574

the systems. The average values obtained are shown in the last line in each table. We compare these values using Friedman.

Regarding NAPFD, Table 3 shows FRRMAB stands out in 68% of the cases (15 out of 22 cases, considering 11 systems and 2 functions). Such cases are represented by grey cells in the table. FRRMAB is followed by UCB with 50% of the cases (11 out of 22). FRRMAB is the best with statistical difference over the others in 8 cases. UCB is the best only in one case for *RNFail*. The effect size results endorse what we observed in the numbers. Besides that, when there is a statistical difference among policies, the effect size tends to be *large*. The performance of these policies is similar for both functions. For *RNFail*, FRRMAB stands out in 7 cases (out of 11), and UCB stands out in 5 (out of 11). For *TimeRank*, FRRMAB stands out in 8, and UCB stands out in 6. Overall, we observe that for *RNFail* the other MAB policies perform better. For this reward function, more policies have values that are close to the best one. Among the MAB

policies, the Random policy has the worst performance. Regarding APFDc we have similar findings. FRRMAB stands out in 72% of the cases (16 out of 22), and UCB in 45% of the cases (10 out of 22).

For the other budgets of 10% and 80%, we observed similar results regarding the performance of the policies and functions. As expected, the values for all policies are better for the budget of 80%, which is less restrictive. Considering NAPFD and all the 33 cases involving the three budgets (11 systems × 3 time budgets), FRRMAB stands out the other policies in ≈ 76% of the cases (25 out of 33) and 82% (27 cases), respectively, for the *RNFail* and *TimeRank* functions. FRRMAB is the best policy with statistical difference in ≈ 40% of the cases (13 out of 33) for both reward functions, and UCB policy is the best in ≈ 6% of the cases (2) and ≈ 3% of the cases (1), respectively, for the *RNFail* and *TimeRank* functions. FRRMAB also obtains the best APFDc values and stands out in ≈ 82% of the cases

(27 out of 33) for the *RNFail* function and in $\approx 76\%$ of the cases (25) for the *TimeRank* function.

For *IOF/ROL*, the approach had the worst performance. This system has a high test case volatility and a high number of peaks of failure detection. In this way, it is an example of that to find a solution to the TCPPI problem is hard. We observe that the Random policy is defeated by MAB policies in almost all systems, except in *ZXing* in both reward functions. To understand this behavior, we analyze the failures detected over the cycles. In this system, we verify peaks in the failure detection in a few commits and long periods without failures. This scenario endorses our approach behavior once that it is based on historical test data. Furthermore, this system has a low test case volatility.

RFTC results are available in supplementary material [28]. We observe that the NAPFD average values found in each independent execution by reward functions and the early fault detection given by RFTC are, in most of the cases, correlated, that is, good NAPFD values provide good RFTC values. In most of the systems, the MAB policies are better than Random policy, as well as they are more stable (low dispersion of values). Among the MAB policies, FRRMAB is more stable than the other ones. This shows that the use of a sliding window is interesting when a system contains peaks of detecting faults and periods of stability, as well as when there is a large number of commits.

RQ1: Results show FRRMAB is the best MAB policy, regarding NAPFD, APFDc values, and early fault detection given by the indicator RFTC. This happens for both reward functions, and the three budgets evaluated. As expected, better indicator values are obtained for the less restrictive budget of 80%. This happens for all policies. In particular, the combination of FRRMAB with TimeRank function provides better performance.

6.2 RQ2: COLEMAN Applicability

To answer RQ2, we use Table 4. The second and third columns presents, respectively, the mean prioritization time and standard deviation of COLEMAN with the FRRMAB policy (the best policy found in the analysis of RQ1), and time budget of 50%. In such a table, we can observe the time spent is negligible in most of the systems. A great time is spent in *Druid* and *Fastjson*, systems that also have a great number of test cases in a CI Cycle. If we take the standard deviation and the worst case (*Fastjson*), COLEMAN takes less than one second to execute. The mean time spent with both reward functions is similar, as well as we do not observe any impact on the other budgets.

However, we need to consider the impact of this time in the complete CI Cycle and the interval between the CI Cycles, and consequently, to observe whether the spent prioritization time is expensive. In addition, it is important to analyze the impact concerning the time spent to reveal the first failure, which can be used to reduce the time spent in a CI Cycle once that the test execution cost can be reduced when a failure is revealed because the test is ended. For this end, Table 5 presents the time reduction average, given by the indicator NTR (Equation 6) with both reward functions, over the three budgets. Refer to Table 2 to see duration of the CI cycles and the interval between them.

Table 4: Mean and standard deviation Prioritization Time (in seconds) with time budget of 50%.

SUT	PRIORITIZATION TIME (SEC.)			
	FRRMAB		ANN	
	RNFail	TimeRank	RNFail	TimeRank
Druid	0.2474 \pm 0.040	0.2373 \pm 0.041	0.3881 \pm 0.268	0.3844 \pm 0.279
Fastjson	0.3916 \pm 0.191	0.3879 \pm 0.191	0.2474 \pm 1.382	0.2395 \pm 1.288
Deeplearning4j	0.0271 \pm 0.002	0.0271 \pm 0.002	0.0038 \pm 0.006	0.0187 \pm 0.107
DSpace	0.0287 \pm 0.002	0.0287 \pm 0.002	0.0101 \pm 0.042	0.0507 \pm 0.186
Guava	0.0609 \pm 0.016	0.0609 \pm 0.016	0.0335 \pm 0.039	0.0405 \pm 0.066
OkHttp	0.0283 \pm 0.002	0.0283 \pm 0.002	0.0079 \pm 0.022	0.0371 \pm 0.125
Retrofit	0.0277 \pm 0.002	0.0277 \pm 0.002	0.0050 \pm 0.010	0.0178 \pm 0.079
ZXing	0.0343 \pm 0.003	0.0343 \pm 0.003	0.0156 \pm 0.025	0.0229 \pm 0.069
IOF/ROL	0.0278 \pm 0.005	0.0278 \pm 0.005	0.0034 \pm 0.009	0.5364 \pm 1.221
Paint Control	0.0256 \pm 0.002	0.0256 \pm 0.002	0.0020 \pm 0.005	0.0028 \pm 0.028
GSDTSR	0.0253 \pm 0.002	0.0253 \pm 0.002	0.0027 \pm 0.014	0.0028 \pm 0.013
Average	0.0841	0.0828	0.0654	0.1231

Table 5 shows percentages of time reduction provided by our approach are close to 73% for *IOF/ROL*, 47% for *Deeplearning4j*, and 43% for *Druid*. This reduction in a CI cycle is due to the early fault detection provided by our approach. But, in most systems, we can see a low percentage. The performance of both functions is similar, as well as the NTR values for each system. In overall, the *RNFail* function obtains better reduction than *TimeRank* considering time budget of 80%, and the *TimeRank* is better in the other time budgets. Comparing the average percentage for the budget of 50% we observe an improvement regarding the budget of 10%. But this difference is slightly lower when we compare the budgets of 50% and 80%. Then, the budget may be a reason to explain low reduction. Other possible reasons are the difficulty inherent to the TCPPI problem and the need to consider other aspects in the prioritization, such as the time to execute each test case.

Regarding the time spent in a CI Cycle and interval between commits for each system, we observe that a new commit is typically performed after a CI Cycle is finished and with a considered time, due to the time between commits is, in most systems, higher than the time spent by a CI Cycle. In this way, the systems chosen do not present a situation with multiple test requests. In addition, the time spent in a CI Cycle and between cycles is in minutes, and as our approach spent, in the worst case, less than one second to prioritize the test cases, there is not a negative impact in the use of our approach.

RQ2: Our approach is applicable to the CI context. It contributes to reducing the time spent in a CI cycle, even with a restrictive budget of 10%. In some cases, the reduction can achieve a percentage of 70%.

6.3 RQ3: Comparing COLEMAN and RETECS

To answer RQ3, we compare COLEMAN using FRRMAB (the best policy according to RQ1) with RETECS using ANN. Table 5 shows the NAPFD and APFDc results obtained.

As we can observe for *RNFail* function and NAPFD, FRRMAB stands out in $\approx 70\%$ of the cases (23 out of 33). ANN stands out in 33% (11 cases). FRRMAB is the best with statistical difference in $\approx 66\%$ (22 out of 33) of the cases against 30% (10 cases) for ANN. Regarding the NAPFD average across the SUTs, FRRMAB obtained the best results, with statistical difference in the time budget of 80%.

Table 5: Mean and standard deviation NAPFD, APFDc, and NTR values: COLEMAN against RETECS.

(See caption of Table 3 for a description of the headings.)

SUT	NAPFD				APFDc				NTR			
	RNFail		TimeRank		RNFail		TimeRank		RNFail		TimeRank	
	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB	ANN	FRRMAB
Time Budget: 10%												
Druid	0.6768 ± 0.129 ▲	0.6801 ± 0.052 ★	0.6488 ± 0.074 ▲	0.7137 ± 0.074 ★	0.8067 ± 0.088 ★	0.6964 ± 0.063 ▲	0.7815 ± 0.051 ★	0.7181 ± 0.076 △	0.1863 ± 0.124	0.2027 ± 0.115	0.1556 ± 0.077	0.2355 ± 0.135
FastJson	0.8714 ± 0.015 ▲	0.9030 ± 0.014 ★	0.8719 ± 0.005 ▲	0.8980 ± 0.018 ★	0.8805 ± 0.014 ▲	0.9064 ± 0.013 ★	0.8310 ± 0.004 ▲	0.8974 ± 0.018 ★	0.0194 ± 0.016	0.3117 ± 0.221	0.0219 ± 0.007	0.2674 ± 0.232
DeepLearning4j	0.6615 ± 0.072 ▲	0.7533 ± 0.002 ★	0.6739 ± 0.016 ▲	0.7716 ± 0.000 ★	0.8135 ± 0.023 ★	0.7766 ± 0.001 ▲	0.8185 ± 0.010 ★	0.7773 ± 0.000 ▲	0.5488 ± 0.029	0.4626 ± 0.001	0.5548 ± 0.015	0.4663 ± 0.000
DSpace	0.9437 ± 0.001 ▲	0.9489 ± 0.003 ★	0.9410 ± 0.001 ▲	0.9496 ± 0.004 ★	0.9480 ± 0.001 ▲	0.9510 ± 0.002 ★	0.9458 ± 0.001 ▲	0.9513 ± 0.004 ★	0.0188 ± 0.001	0.0370 ± 0.015	0.0105 ± 0.001	0.0393 ± 0.018
Guava	0.9676 ± 0.015 ★	0.9554 ± 0.002 ▲	0.9563 ± 0.004	0.9586 ± 0.001 ★	0.9811 ± 0.007 ★	0.9561 ± 0.001 ▲	0.9761 ± 0.003 ★	0.9582 ± 0.001 ▲	0.0449 ± 0.015	0.0471 ± 0.016	0.0367 ± 0.005	0.0492 ± 0.015
OkHttp	0.8357 ± 0.002 ★	0.8323 ± 0.000 ▲	0.8095 ± 0.006 ▲	0.8407 ± 0.000 ★	0.8484 ± 0.001 ★	0.8378 ± 0.000 ▲	0.8292 ± 0.006 ▲	0.8425 ± 0.000 ★	0.0830 ± 0.001	0.0658 ± 0.000	0.0579 ± 0.008	0.0702 ± 0.000
Retrofit	0.9641 ± 0.001 ★	0.9639 ± 0.000	0.9621 ± 0.001 ▲	0.9642 ± 0.000 ★	0.9672 ± 0.001 ★	0.9646 ± 0.000 ▲	0.9655 ± 0.001 ★	0.9648 ± 0.000 △	0.0088 ± 0.000	0.0070 ± 0.000	0.0076 ± 0.001	0.0073 ± 0.000
Zxing	0.9854 ± 0.000 ★	0.9826 ± 0.000 ▲	0.9855 ± 0.000 ★	0.9828 ± 0.000 ▲	0.9893 ± 0.000 ★	0.9832 ± 0.000 ▲	0.9893 ± 0.000 ★	0.9835 ± 0.000 ▲	0.0122 ± 0.000	0.0037 ± 0.000	0.0126 ± 0.001	0.0037 ± 0.000
IOF/ROL	0.3704 ± 0.005 ★	0.3632 ± 0.001 ▲	0.3779 ± 0.003 ★	0.3670 ± 0.001 ▲	0.3746 ± 0.005 ★	0.3661 ± 0.001 ▲	0.3819 ± 0.003 ★	0.3701 ± 0.001 ▲	0.5133 ± 0.030	0.5585 ± 0.007	0.5646 ± 0.015	0.5701 ± 0.004
Paint Control	0.9078 ± 0.000 ★	0.9076 ± 0.000 ▲	0.9077 ± 0.000 ★	0.9076 ± 0.000 ▲	0.9082 ± 0.000 ★	0.9080 ± 0.000 ▲	0.9081 ± 0.000 ★	0.9080 ± 0.000 ▲	0.1151 ± 0.000	0.1133 ± 0.000	0.1139 ± 0.000	0.1131 ± 0.000
GSDTSR	0.9893 ± 0.000 ▲	0.9894 ± 0.000 ★	0.9893 ± 0.000 ▲	0.9894 ± 0.000 ★	0.9894 ± 0.000 ★	0.9894 ± 0.000 ★	0.9894 ± 0.000 ★	0.9894 ± 0.000 △	0.0087 ± 0.000	0.0093 ± 0.000	0.0090 ± 0.000	0.0096 ± 0.000
Average	0.8340	0.8436	0.8294	0.8494	0.8643	0.8487	0.8294	0.8494	0.1428	0.1653	0.1404	0.1665
Time Budget: 50%												
Druid	0.6851 ± 0.134 ▲	0.9333 ± 0.013 ★	0.6323 ± 0.074 ▲	0.9710 ± 0.008 ★	0.8147 ± 0.102 ▲	0.9486 ± 0.016 ★	0.7597 ± 0.051 ▲	0.9787 ± 0.009 ★	0.1840 ± 0.137	0.4057 ± 0.013	0.1213 ± 0.071	0.4225 ± 0.008
FastJson	0.8714 ± 0.007 ▲	0.9174 ± 0.021 ★	0.8902 ± 0.013 ▲	0.9118 ± 0.028 ★	0.9326 ± 0.005 ★	0.9186 ± 0.021 ▲	0.9392 ± 0.017 ★	0.9140 ± 0.027 ▲	0.0399 ± 0.010	0.3539 ± 0.262	0.0602 ± 0.020	0.3394 ± 0.282
DeepLearning4j	0.7049 ± 0.007 ▲	0.7890 ± 0.001 ★	0.6562 ± 0.018 ▲	0.8200 ± 0.000 ★	0.8331 ± 0.041 ★	0.8106 ± 0.001 ▲	0.8379 ± 0.012 ★	0.8134 ± 0.001 ▲	0.5276 ± 0.039	0.4695 ± 0.000	0.5447 ± 0.010	0.4625 ± 0.000
DSpace	0.9568 ± 0.001 ▲	0.9724 ± 0.009 ★	0.9485 ± 0.001 ▲	0.9766 ± 0.008 ★	0.9683 ± 0.001	0.9737 ± 0.009 ★	0.9615 ± 0.001 ▲	0.9767 ± 0.009 ★	0.0334 ± 0.001	0.0751 ± 0.031	0.0218 ± 0.002	0.0776 ± 0.034
Guava	0.9502 ± 0.015 ▲	0.9553 ± 0.004 ★	0.9578 ± 0.004 ▲	0.9675 ± 0.007 ★	0.9675 ± 0.008 ★	0.9687 ± 0.003 ▲	0.9806 ± 0.005 ★	0.9672 ± 0.007 ▲	0.0303 ± 0.016	0.0662 ± 0.024	0.0387 ± 0.006	0.0659 ± 0.020
OkHttp	0.8812 ± 0.010 ▲	0.9192 ± 0.000 ★	0.8446 ± 0.003 ▲	0.9317 ± 0.000 ★	0.8578 ± 0.015 ▲	0.9177 ± 0.000 ★	0.8869 ± 0.002 ▲	0.9246 ± 0.000 ★	0.1118 ± 0.014	0.1431 ± 0.000	0.1060 ± 0.003	0.1486 ± 0.000
Retrofit	0.9706 ± 0.002 ▲	0.9853 ± 0.000 ★	0.9718 ± 0.002 ▲	0.9893 ± 0.000 ★	0.9762 ± 0.002 ▲	0.9850 ± 0.000 ★	0.9778 ± 0.002 ▲	0.9885 ± 0.000 ★	0.0134 ± 0.001	0.0156 ± 0.000	0.0138 ± 0.001	0.0172 ± 0.000
Zxing	0.9878 ± 0.000 ★	0.9846 ± 0.000 ▲	0.9881 ± 0.001 ★	0.9857 ± 0.000 ▲	0.9954 ± 0.000 ★	0.9862 ± 0.000 ▲	0.9956 ± 0.001 ★	0.9869 ± 0.000 ▲	0.0201 ± 0.000	0.0109 ± 0.000	0.0201 ± 0.001	0.0110 ± 0.000
IOF/ROL	0.5101 ± 0.007 ★	0.5046 ± 0.002 ▲	0.5025 ± 0.006 ▲	0.5189 ± 0.002 ★	0.5175 ± 0.008 ★	0.5081 ± 0.002 ▲	0.5043 ± 0.006 ▲	0.5223 ± 0.002 ★	0.7037 ± 0.019	0.7110 ± 0.003	0.6834 ± 0.014	0.7193 ± 0.003
Paint Control	0.9150 ± 0.000 ★	0.9150 ± 0.000 ▲	0.9138 ± 0.000 ▲	0.9150 ± 0.000 ★	0.9171 ± 0.000 ★	0.9162 ± 0.000 ▲	0.9162 ± 0.000 ★	0.9162 ± 0.000 ★	0.1283 ± 0.000	0.1222 ± 0.000	0.1142 ± 0.001	0.1223 ± 0.000
GSDTSR	0.9911 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9906 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9911 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9910 ± 0.000 ★	0.9894 ± 0.000 ▲	0.0199 ± 0.000	0.0093 ± 0.000	0.0179 ± 0.000	0.0096 ± 0.000
Average	0.8567	0.8978	0.8451	0.9070	0.8919	0.9021	0.8862	0.9071	0.1648	0.2166	0.1583	0.2178
Time Budget: 80%												
Druid	0.6490 ± 0.113 ▲	0.9380 ± 0.012 ★	0.6551 ± 0.099 ▲	0.9830 ± 0.003 ★	0.7142 ± 0.111 ▲	0.9469 ± 0.015 ★	0.6881 ± 0.090 ▲	0.9912 ± 0.004 ★	0.1477 ± 0.113	0.4069 ± 0.014	0.1230 ± 0.096	0.4292 ± 0.004
FastJson	0.8708 ± 0.007 ▲	0.9536 ± 0.010 ★	0.8925 ± 0.010 ▲	0.9242 ± 0.028 ★	0.9037 ± 0.008 ▲	0.9488 ± 0.012 ★	0.9133 ± 0.015	0.927 ± 0.026 ★	0.0385 ± 0.011	0.4613 ± 0.278	0.0516 ± 0.018	0.3963 ± 0.301
DeepLearning4j	0.7058 ± 0.091 ▲	0.8424 ± 0.001 ★	0.6640 ± 0.016 ▲	0.8641 ± 0.001 ★	0.8158 ± 0.064 ★	0.8068 ± 0.002 ▼	0.8522 ± 0.012 ★	0.7989 ± 0.001 ▲	0.5016 ± 0.058	0.4224 ± 0.001	0.5475 ± 0.007	0.4047 ± 0.000
DSpace	0.9601 ± 0.001 ▲	0.9792 ± 0.006 ★	0.9508 ± 0.001 ▲	0.9825 ± 0.007 ★	0.9738 ± 0.001 △	0.9796 ± 0.006 ★	0.9639 ± 0.001 ▲	0.9810 ± 0.008 ★	0.0374 ± 0.001	0.0800 ± 0.031	0.0269 ± 0.002	0.0812 ± 0.034
Guava	0.9441 ± 0.012 ▲	0.9784 ± 0.012 ★	0.9581 ± 0.007 ▲	0.9841 ± 0.014 ★	0.9627 ± 0.010 ▲	0.9780 ± 0.013 ★	0.9689 ± 0.009 △	0.9825 ± 0.015 ★	0.0247 ± 0.013	0.0812 ± 0.044	0.0348 ± 0.009	0.0888 ± 0.045
OkHttp	0.9027 ± 0.013 ▲	0.9350 ± 0.000 ★	0.8558 ± 0.004 ▲	0.9478 ± 0.000 ★	0.8836 ± 0.020 ▲	0.9271 ± 0.000 ★	0.8974 ± 0.003 ▲	0.9462 ± 0.000 ★	0.1112 ± 0.017	0.1493 ± 0.000	0.1153 ± 0.003	0.1351 ± 0.000
Retrofit	0.9724 ± 0.005 ▲	0.9881 ± 0.000 ★	0.9745 ± 0.003 ▲	0.9916 ± 0.000 ★	0.9785 ± 0.004 ▲	0.9873 ± 0.000 ★	0.9808 ± 0.003 ▲	0.9903 ± 0.000 ★	0.0140 ± 0.001	0.0161 ± 0.000	0.0145 ± 0.001	0.0179 ± 0.000
Zxing	0.9878 ± 0.000 ▲	0.9972 ± 0.000 ★	0.9883 ± 0.001 ▲	0.9996 ± 0.000 ★	0.9953 ± 0.000 ▲	0.9975 ± 0.000 ★	0.9953 ± 0.001 ▲	0.9996 ± 0.000 ★	0.0201 ± 0.000	0.0224 ± 0.000	0.0197 ± 0.002	0.0227 ± 0.000
IOF/ROL	0.5495 ± 0.006 ▲	0.5569 ± 0.002 ★	0.5287 ± 0.007 ▲	0.5678 ± 0.001 ★	0.5593 ± 0.006 ★	0.5591 ± 0.002	0.5311 ± 0.006 ▲	0.5699 ± 0.001 ★	0.7263 ± 0.015	0.7293 ± 0.002	0.6857 ± 0.011	0.7334 ± 0.001
Paint Control	0.9162 ± 0.000 ▲	0.9171 ± 0.000 ★	0.9160 ± 0.000 ▲	0.9171 ± 0.000 ★	0.9187 ± 0.000 ★	0.9176 ± 0.000 ▲	0.9158 ± 0.000 ▲	0.9177 ± 0.000 ★	0.1285 ± 0.000	0.1209 ± 0.000	0.1161 ± 0.001	0.1204 ± 0.000
GSDTSR	0.9921 ± 0.000 ★	0.9893 ± 0.000 ▲	0.9914 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9919 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9917 ± 0.000 ★	0.9894 ± 0.000 ▲	0.0218 ± 0.001	0.0093 ± 0.000	0.0203 ± 0.000	0.0096 ± 0.000
Average	0.8591	0.9159	0.8523	0.9228	0.8816	0.9126	0.8817	0.9167	0.1611	0.2272	0.1596	0.2236

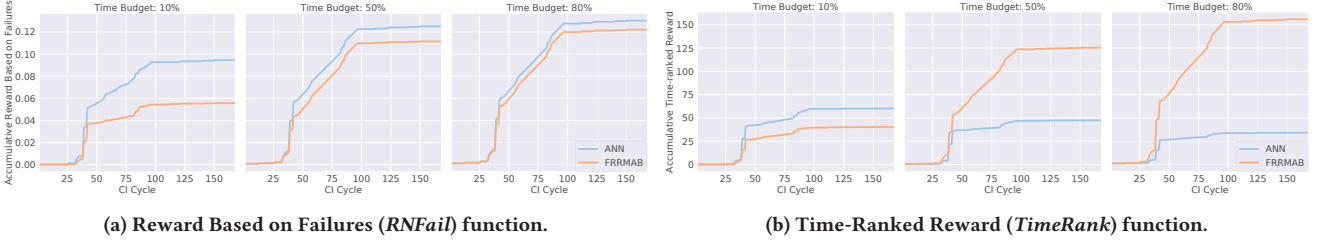


Figure 3: Accumulative Reward values over the CI Cycles from Druid system.

With the *TimeRank* function, FRRMAB presents even better results. FRRMAB stands out in $\approx 82\%$ (27 out of 33) of the cases against 21% (7 cases) for ANN. FRRMAB is the best with statistical difference in $\approx 79\%$ of the cases (26 out of 33) against 18% (6 cases) for ANN. Across the SUTs, FRRMAB is the best, with statistical difference considering the time budgets of 50% and 80%.

We observe that ANN has a bad performance for Druid and DeepLearning4j, and improves performance in comparison with FRRMAB when the budget decreases. On the other hand, FRRMAB presents more stability in NAPFD average values than ANN. FRRMAB has the best performance with low variation and better values for Druid and DeepLearning4j. Regarding APFDc values, ANN using *RNFail* function stands out in 64% of the cases (21 out of 33) against 45% (15 cases) for FRRMAB. ANN is the best with statistical difference in $\approx 55\%$ (18 out of 33) of the cases against

30% (10) for FRRMAB. Considering *TimeRank* the opposite occurs. FRRMAB stands out in 55% of the cases (18 out of 33), against 48% (16) for ANN. FRRMAB is the best with statistical difference in $\approx 52\%$ (17 out of 33) of the cases against 42% (14) for ANN. Across the SUTs, overall, FRRMAB and ANN are statistically equivalent in most of the time budgets and reward functions, except in *TimeRank* function with a time budget of 80% in which FRRMAB is better than ANN. For this indicator, the results show the approaches are competitive, that is, there is not a great difference between them. A possible reason for this is that RETECS considers the individual test case duration, and COLEMAN, differently, considers only historical failure data.

FRRMAB with *TimeRank* produces better values concerning NAPFD, APFDc, and time budgets of 50% and 80%. But overall, the results of both functions are similar.

Table 6: Mean and standard deviation RFTC values: COLEMAN against RETECS.

SUT	RFTC			
	RNFail		TimeRank	
	ANN	FRRMAB	ANN	FRRMAB
TIME BUDGET: 10%				
Druid	1166.4411 ± 546.049 ▲	209.3289 ± 98.987 ★	1240.8713 ± 356.817 ▲	56.1579 ± 31.056 ★
Fastjson	1094.1147 ± 209.106 ▲	184.9911 ± 65.066 ★	998.7949 ± 177.738 ▲	96.0378 ± 36.22 ★
Deeplearning4j	5.3151 ± 2.456 ▲	2.3049 ± 0.048 ★	5.7919 ± 0.677 ★	2.0437 ± 0.007 ★
Dspace	11.9561 ± 0.753 ▲	3.024 ± 1.33 ★	13.6276 ± 0.754 ▲	2.1428 ± 0.67 ★
Guava	129.3477 ± 99.443 ▲	31.8991 ± 20.546 ★	191.1219 ± 29.545 ▲	15.0977 ± 13.75 ★
OkHttp	7.4112 ± 0.397 ▲	4.3424 ± 0.0 ★	15.8935 ± 1.951 ▲	1.8039 ± 0.0 ★
Retrofit	3.7352 ± 0.439 ▲	1.5152 ± 0.0 ★	4.7297 ± 0.772 ▲	1.7941 ± 0.0 ★
Zxing	39.7929 ± 0.643 ▲	1.0 ± 0.0 ★	39.1204 ± 1.701 ▲	1.0 ± 0.0 ★
IOF/ROL	1.6445 ± 0.158 ▲	1.2626 ± 0.064 ★	1.5639 ± 0.108 ▲	1.0992 ± 0.05 ★
Paint Control	1 ± 0.0	1 ± 0.0	1.001 ± 0.002 ▽	1.0 ± 0.0 ★
GSDTSR	3.2553 ± 0.387 ▲	2.1316 ± 0.1 ★	3.8958 ± 0.18 ▲	1.1482 ± 0.071 ★
TIME BUDGET: 50%				
Druid	1225.6197 ± 600.746 ▲	121.8396 ± 43.763 ★	1420.3143 ± 418.926 ▲	51.2697 ± 11.926 ★
Fastjson	1527.9434 ± 93.004 ▲	315.8923 ± 79.836 ★	1173.9871 ± 321.789 ▲	335.9929 ± 125.629 ★
Deeplearning4j	5.0267 ± 1.827 ▲	2.5496 ± 0.011 ★	7.3264 ± 0.530 ▲	2.464 ± 0.005 ★
Dspace	19.0354 ± 0.887 ▲	5.5312 ± 1.939 ★	27.6301 ± 0.921 ▲	4.1089 ± 1.769 ★
Guava	289.1386 ± 99.054 ▲	78.6409 ± 45.928 ★	235.0919 ± 27.865 ▲	24.0869 ± 21.479 ★
OkHttp	4.203 ± 2.066 ▲	4.188 ± 0.014 ★	19.6306 ± 0.79 ▲	2.3643 ± 0.002 ★
Retrofit	5.4302 ± 0.43 ▲	2.4059 ± 0.0 ★	5.625 ± 0.415 ▲	1.4299 ± 0.0 ★
Zxing	51.2576 ± 0.579 ▲	5.6 ± 0.0 ★	49.4232 ± 3.15 ▲	2.282 ± 0.0 ★
IOF/ROL	1.8009 ± 0.292 ▲	1.2588 ± 0.035 ★	1.9213 ± 0.218 ▲	1.151 ± 0.024 ★
Paint Control	1.0234 ± 0.004 ▲	1.0018 ± 0.001 ★	1.0257 ± 0.007 ▲	1.0014 ± 0.001 ★
GSDTSR	1.9072 ± 0.06 ★	2.1461 ± 0.101 ▲	3.5648 ± 0.141 ▲	1.1505 ± 0.072 ★
TIME BUDGET: 80%				
Druid	1427.8103 ± 493.429 ▲	146.9112 ± 46.436 ★	1035.3369 ± 658.042 ▲	50.3805 ± 11.25 ★
Fastjson	1535.2998 ± 101.625 ▲	398.4345 ± 105.27 ★	993.382 ± 393.441 ▲	572.0954 ± 221.573 ★
Deeplearning4j	5.5748 ± 2.358 ▲	2.8146 ± 0.014 ★	7.8533 ± 0.581 ▲	2.5017 ± 0.011 ★
Dspace	23.126 ± 1.021 ▲	6.8651 ± 2.946 ★	33.4617 ± 1.119 ▲	6.0794 ± 3.343 ★
Guava	330.5342 ± 74.0 ▲	84.6856 ± 34.075 ★	202.2301 ± 47.258 ▲	83.9989 ± 78.446 ★
OkHttp	4.2988 ± 2.242 ▲	4.0748 ± 0.021 ▲	21.5784 ± 1.148 ▲	2.282 ± 0.003 ★
Retrofit	5.9252 ± 0.884 ▲	2.4636 ± 0.0 ★	5.8832 ± 0.587 ▲	1.4386 ± 0.0 ★
Zxing	51.0061 ± 1.233 ▲	4.2727 ± 0.0 ★	48.6848 ± 2.926 ▲	1.3636 ± 0.0 ★
IOF/ROL	2.021 ± 0.447 ▲	1.317 ± 0.026 ★	2.5248 ± 0.362 ▲	1.2366 ± 0.017 ★
Paint Control	1.015 ± 0.002 ▲	1.0003 ± 0.001 ★	1.0344 ± 0.009 ▲	1.0003 ± 0.001 ★
GSDTSR	1.9413 ± 0.322 ★	2.1461 ± 0.101 ▲	3.4858 ± 0.112 ▲	1.1505 ± 0.072 ★

According to Table 4, ANN is a bit faster than FRRMAB considering the function *RNFail*, and slower considering *TimeRank*. FRRMAB is stable for both functions, whilst ANN spends more time using *TimeRank* function. Analyzing the prioritization time spent along with CI Cycles, we observe that both approaches spend more time when test cases are added or removed from one cycle to another. This occurs because both approaches need to update the information about the test cases, mainly when a high number of test cases increases or decreases abruptly. But a greater impact is observed for RETECS. In particular, ANN has a great variation in the spent time for IOF/ROL. For this system, RETECS can take in the worst case more than one second.

Regarding NTR, for the three budgets, FRRMAB is the best in $\approx 64\%$ (21 out of 33) of the cases against 36% (12) for ANN for *RNFail* function. Considering *TimeRank*, FRRMAB is the best in $\approx 73\%$ (24) of the cases against $\approx 27\%$ (9) for ANN. Regarding RFTC (Table 6) and *RNFail*, FRRMAB stands out in $\approx 94\%$ of the cases (31 out of 33) against $\approx 12\%$ (4) for ANN. But in 2 of these 4 cases, ANN and FRRMAB are statistically equivalent. With *TimeRank*, FRRMAB is the best in all cases.

With the systems used, we do not identify a scalability pattern concerning the time spent to prioritize the test cases. The use of systems with a great number of test cases in each CI Cycles can help in the identification of our approach scalability in future studies.

Due to the bad performance of ANN in Druid, we analyze this system. Druid has the lowest number of CI Cycles (168) among the systems evaluated, as well as it has many tests (2391). Figure 3 shows the accumulative reward values from ANN (blue line) and FRRMAB (orange line) along the CI Cycles considering *RNFail* and

TimeRank functions. We can observe that although RETECS better assigns the rewards in the *RNFail* function, the prioritization order is not adequate, which we can see with the *TimeRank* function. In this way, FRRMAB better mitigates than ANN the problem of beginning without learning mainly using the *TimeRank* function, and adapts quickly to deal with a peak of faults in the first cycles⁵.

RQ3: COLEMAN using FRRMAB outperforms RETECS using ANN regarding NAPFD, RFTC and NTR indicators, independently of the reward functions and budgets investigated. Regarding APFDc, both approaches present similar results. Besides, with respect to the execution time, FRRMAB is more stable, that is, adapts better to deal with peak of faults.

7 CONCLUDING REMARKS

This work introduces COLEMAN, an approach to deal with the TCPCI problem. COLEMAN is based on MAB with combinatorial and volatile characteristics to dynamically obtain an adequate prioritized test suite for each CI Cycle (commit) using historical failure data of test cases. The approach properly deals with the EvE dilemma and takes into account TCPCI characteristics such as test case volatility.

We evaluated our approach concerning three time budgets: 10%, 50%, and 80%; with five MAB policies: FRRMAB, UCB, ϵ -Greedy, Greedy, and Random; and compared it against an RL-based approach from literature, named RETECS [33]. The results show that FRRMAB policy outperforms the other policies with both reward functions. In most cases, our approach outperforms RETECS in terms of NAPFD, NTR, and earlier fault detection (RFTC). It also does not present great variations in the prioritization time, and does not require any additional information. Regarding APFDc, the obtained values are competitive with RETECS.

Our approach spends, in the worst case, less than one second to prioritize a test suite, and the obtained results in many aspects investigated show high performance. Then, we can conclude that COLEMAN contributes efficiently and effectively to address the TCPCI problem.

Future work includes the application of our approach in other systems, possibly in an industrial scenario, as well as the study of other policies to include the individual test case duration in the prioritization. We intend to evaluate the scalability of our approach and provide a relationship between the number of test cases and the prioritization time. Another work to be conducted is to provide COLEMAN as an API or an *add-on*, as well as the source code.

ACKNOWLEDGMENTS

This work is supported by the Brazilian funding agencies CAPES and CNPq. Grant: 305968/2018.

REFERENCES

- [1] Venkatachalam Anantharam, Pravin Varaiya, and Jean Walrand. 1987. Asymptotically Efficient Allocation Rules for the Multiarmed Bandit Problem with Multiple Plays-Part I: I.I.D. Rewards. *IEEE Trans. Automat. Control* 32, 11 (November 1987), 968–976. <https://doi.org/10.1109/TAC.1987.1104491>
- [2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2 (01 May 2002), 235–256. <https://doi.org/10.1023/A:1013689704352>

⁵See supplementary material about characteristics of this system.

- [3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.
- [4] J. M. Benitez, J. L. Castro, and I. Requena. 1997. Are Artificial Neural Networks Black Boxes? *Transactions on Neural Networks* 8, 5 (Sept. 1997), 1156–1164. <https://doi.org/10.1109/72.623216>
- [5] Zahy Bnaya, Rami Puzis, Roni Stern, and Ariel Felner. 2013. Volatile Multi-Armed Bandits for Guaranteed Targeted Social Crawling. In *Proceeding of Workshops at the 27th AAAI Conference on Artificial Intelligence (Late-Breaking Developments)*. 8–10.
- [6] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 975–980. <https://doi.org/10.1145/2950290.2983954>
- [7] Catagay Catal and Deepti Mishra. 2013. Test case prioritization: a systematic mapping study. *Software Quality Journal* 21, 3 (01 Sep 2013), 445–478. <https://doi.org/10.1007/s11219-012-9181-z>
- [8] Y. Cho, J. Kim, and E. Lee. 2016. History-Based Test Case Prioritization for Failure Information. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 385–388. <https://doi.org/10.1109/APSEC.2016.066>
- [9] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. 2008. An Empirical Study of the Effect of Time Constraints on the Cost-benefits of Regression Testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 71–82. <https://doi.org/10.1145/1453101.1453113>
- [10] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
- [11] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*. 329–338. <https://doi.org/10.1109/ICSE.2001.919106>
- [12] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 235–245. <https://doi.org/10.1145/2635868.2635910>
- [13] A. Fialho. 2010. *Adaptive operator selection for optimization*. Ph.D. Dissertation. Université Paris Sud-Paris XI.
- [14] M. Friedman. 1940. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics* 11, 1 (1940), 86–92.
- [15] Alireza Haghighathkhan, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98. <https://doi.org/10.1016/j.jss.2018.08.061>
- [16] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-box and Black-box Test Prioritization. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 523–534. <https://doi.org/10.1145/2884781.2884791>
- [17] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 426–437. <https://doi.org/10.1145/2970276.2970358>
- [18] Muhammad Khatibsyarhini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93. <https://doi.org/10.1016/j.infsof.2017.08.014>
- [19] W. H. Kruskal and W. A. Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621.
- [20] Volodymyr Kuleshov and Doina Precup. 2014. Algorithms for multi-armed bandit problems. *Journal of Machine Learning Research* 1 (2014), 1–48.
- [21] K. Li, A. Fialho, S. Kwong, and Q. Zhang. 2014. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on* 18, 1 (2014), 114–130.
- [22] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining Prioritization: Continuous Prioritization for Continuous Integration. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 688–698. <https://doi.org/10.1145/3180155.3180213>
- [23] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.
- [24] Dusica Marijan. 2015. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*. IEEE Computer Society, 157–162. <https://doi.org/10.1109/QRS.2015.31>
- [25] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *2013 IEEE International Conference on Software Maintenance*. 540–543. <https://doi.org/10.1109/ICSM.2013.91>
- [26] D. Marijan, M. Liaen, A. Gotlieb, S. Sen, and C. Ieva. 2017. TITAN: Test Suite Optimization for Highly Configurable Software. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 524–531. <https://doi.org/10.1109/ICST.2017.60>
- [27] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 233–242. <https://doi.org/10.1109/ICSE-SEIP.2017.16>
- [28] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration environments. <https://doi.org/10.17605/OSF.IO/WMCBT>
- [29] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020), 106268. <https://doi.org/10.1016/j.infsof.2020.106268>
- [30] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. 2007. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *IEEE International Conference on Software Maintenance*. 255–264. <https://doi.org/10.1109/ICSM.2007.4362638>
- [31] Herbert Robbins. 1985. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*. Springer, 169–177.
- [32] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*. IEEE Computer Society, 179–.
- [33] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 12–22. <https://doi.org/10.1145/3092703.3092709>
- [34] Richard S. Sutton and Andrew G. Barto. 2011. Reinforcement learning: An introduction. *The MIT Press* (2011).
- [35] Travis CI. [n.d.]. Travis CI. <https://travis-ci.org>. Accessed: 2018-01-22.
- [36] Andras Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (Jan. 2000), 101–132.
- [37] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (March 2012), 67–120. <https://doi.org/10.1002/stv.430>
- [38] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. 2017. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 60–71. <https://doi.org/10.1109/ASE.2017.8115619>
- [39] Y. Zhu, E. Shihab, and P. C. Rigby. 2018. Test Re-Prioritization in Continuous Testing Environments. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 69–79. <https://doi.org/10.1109/ICSME.2018.00016>
- [40] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *5th International Conference on Learning Representations (ICLR)*.

APPENDIX C – MULTI-ARMED BANDIT TCPCI: A TRADE-OFF ANALYSIS

Multi-Armed Bandit Test Case Prioritization in Continuous Integration Environments: A Trade-off Analysis

Jackson A. Prado Lima
Department of Informatics
Federal University of Paraná (UFPR)
Curitiba, Paraná, Brazil
japlima@inf.ufpr.br

Silvia R. Vergilio
Department of Informatics
Federal University of Paraná (UFPR)
Curitiba, Paraná, Brazil
silvia@inf.ufpr.br

ABSTRACT

Continuous Integration (CI) practices lead the software to be integrated and tested many times a day, usually subject to a test budget. To deal with this scenario, cost-effective test case prioritization techniques are required. COLEMAN is a Multi-Armed Bandit approach that learns from the test case failure-history the best prioritization order to maximize early fault detection. Reported results show that COLEMAN has reached promising results with different test budgets and spends, in the worst case, less than one second to execute. However, COLEMAN has not been evaluated against a search-based approach. Such an approach can generate near-optimal solutions but is not suitable to the CI budget because it takes too long to execute. Considering this fact, this paper analyses the trade-offs of the COLEMAN solutions in comparison with the near-optimal solutions generated by a Genetic Algorithm (GA). We use measures, which better fit with time constraints: *Normalized Average Percentage of Faults Detected* (NAPFD), *Root-Mean-Square-Error* (RMSE), and *Prioritization Time*. We use seven large-scale real-world software systems, and three different test budgets, 10%, 50%, and 80% of the total time required to execute the test set available for a CI cycle. COLEMAN obtains solutions near to the GA solutions in 90% of the cases, but scenarios with high volatility of test cases and a small number of cycles hamper the prioritization.

KEYWORDS

Test Case Prioritization, Continuous Integration environments, Multi-Armed Bandit

1 INTRODUCTION

In the Continuous Integration (CI) process, developers merge their changes into a shared version control repository after every small task is completed. This practice has become popular because it allows early bug fixes, avoids conflicts, and duplicate efforts. But CI leads the software to be integrated and tested many times a day, and as a consequence increases testing costs. This is because a large test case set may take many minutes, several hours, or even days to execute [5].

The CI environments have other particularities. The constant code changes also lead to many modifications in the test sets available for a build. Then, over the cycles test cases became obsolete and are removed from the set. Others are included and some others can reappear; a fact known as test case volatility. Another particularity of CI environment is that, many times, a time constraint exists to perform a CI cycle, the called test budget, because many projects

share the same environment in the organization. In addition to this, it is very important to provide rapid test feedback.

In this scenario, Regression Testing (RT) plays an important role to make sure that continuously integrated changes in the code should not affect existing functionalities. The literature reports three main techniques for RT, usually applied to reduce time and costs: test case minimization, test case selection, and test case prioritization [24]. Test suite minimization seeks to eliminate redundant test cases following certain objectives. Test case selection seeks to identify the most relevant test cases according to some criteria or recent changes. Test Case Prioritization (TCP) seeks to establish a test case execution order, generally maximizing early fault detection. TCP techniques do not remove test cases from the test set and allows fault-proneness test cases to be executed first, providing rapid feedback on failures in the CI cycle. Because of this, TCP in CI environments (TCPCI) has gained importance and is considered a trendy research topic. A recent map shows that most approaches have been proposed in the last years [17].

Most of the existing approaches are based on the test failure history and try to execute first the test cases that have previously failed, assuming that they are likely to fail again [1, 5, 11–14, 20, 23]. On the other hand, search-based techniques, which are commonly used for TCP in the literature [9], are few explored because they usually take too long to execute, what makes them unsuitable in the presence of the CI time constraints. In fact, approaches that rely on code analysis and instrumentation, can be time-consuming and produce results that quickly become inaccurate due to the frequent changes [3].

We can see the TCPCI problem requires approaches that consider the dynamic characteristics of the CI environments, the test case volatility, and the test budget. In the literature, an approach which successfully addresses all these CI particularities is COLEMAN (*Combinatorial VOLatile Multi-Armed BANDit*) [15]. COLEMAN uses Multi-Armed Bandit (MAB) techniques [7] to learn from past test executions and deal with the EvE (Exploration versus Exploitation) dilemma [19]. In this way, the approach is adaptive and deals with test cases volatility balancing the quantity of error-prone test cases (Exploitation) and the diversity of the test set, given the chance for all test cases, including new ones, to execute (Exploration). COLEMAN can be considered the state-of-the-art approach for TCPCI. Reported results show that COLEMAN overcomes similar approaches in terms of early fault detection and execution time. It takes in all cases less than one second to execute, even for the systems with the greatest number of builds and test cases.

Despite these promising results and advantages, there is no study comparing COLEMAN with near-optimal solutions such as the ones

generated by a search-based approach. Given the fact that search-based approaches consume much time to execute, then a question that arises is: "How far are the solutions produced by a learning approach like COLEMAN from the solutions produced by an optimization approach"?

This paper aims to answer this question reporting experiments results and analyzing the trade-offs of the solutions generated by COLEMAN, regarding fault detection and prioritization time, in comparison with the near-optimal solutions generated by a Genetic Algorithm (GA). In the experiments, we used three measures, which better fits with time constraints: *Normalized Average Percentage of Faults Detected* (NAPFD), *Root-Mean-Square-Error* (RMSE), and *Prioritization Time*. We use seven large-scale real-world software systems, and three different test budgets, considering that the time available for a CI cycle corresponds to respectively 10%, 50% and 80% of the total time required to execute the test case set available for that cycle.

The results show that, except for one system, COLEMAN solutions are very near (or near) to the solutions produced by the GA, and in return, there is a considerable gain in the time, mainly for the hard cases. In this way, the main contribution of this paper is to conduct a trade-off analysis of an on-line learning approach, considered the state-of-the-art in the TCPCI problem. The analysis points out research directions for TCPCI considering the drawbacks and strengths found. Besides that, we provide supplementary material with the source code of the Genetic Algorithm, additional analysis, and results. This replication package allows future investigations [16].

The remainder of this paper is as follows. Section 2 describes COLEMAN. Section 3 presents aspects of our GA implementation. Section 4 details the methodology adopted in our experiments. Section 5 shows and analyses the results. Section 6 reviews related work. Section 7 contains our final remarks and discusses future work.

2 EVALUATED APPROACH

In this section, we describe the TCPCI approach evaluated in our work, COLEMAN [15]. Figure 1 presents how such an approach interacts with the CI environment.

CI environments automate the process of building and testing software, and allow developers to merge code under development or maintenance with the mainline codebase at frequent time intervals. In CI development, teams work continuously integrating code and make smaller code commits every day, usually monitored by a CI source control server. When a change occurs, the CI server clones this code, builds it, and runs the testing process. When the entire process ends, the CI server generates a report (feedback).

Given a test case set T , available for a build, the set PT of all possible permutations of T , and a function f that determines the performance of a given prioritization T' from PT to real numbers, the TCPCI problem aims at finding the best T' to achieve certain specific criteria measured by f . In CI, the determination of T' may subject to a test budget that is the available time to execute the CI cycle.

COLEMAN formulates the TCPCI problem as a *Multi-Armed Bandit* (MAB) problem. MAB problems [7] are sequential decision problems

related to the Exploration versus Exploitation (EvE) dilemma [19]. In such a dilemma, there is a balance in the search between solutions with the best performance and dissimilar solutions. In the MAB scenario, a player plays on a set of slot machines (or arms/actions) that even identical produce different gains. After a player pulls one of the arms in a turn (c), a reward is received drawn from some unknown distribution, thus aiming to maximize the sum of the rewards. Different strategies, called MAB policies, can be used to choose the next arm by observing previous rewards and decisions.

Similarly, COLEMAN considers that a test case is an arm, but it encompasses the dynamic nature of the TCPCI problem. For this, COLEMAN incorporates two variants of MAB: volatile and combinatorial. In the first variation, the approach selects multiple arms in each turn (commit), rather than one, to produce an ordered set. In the second one, only the test cases available in each commit are considered for prioritization. The second variation aims to deal with the test case volatility, in which the test cases (arms) may change dynamically over time. In the end, a reward function is used to obtain feedback (reward) from the prioritization proposed by the approach. Based on such feedback, the approach aims to incorporate the learning from the application of the prioritized test case set. COLEMAN is generic and lightweight. That is, it does not require any further detail about the system under tests such as code coverage or code instrumentation, as well as, it allows the use of any MAB policy and requires only historical failure data.

COLEMAN works with the CI environments according to the following steps (see Figure 1). After a successful build, during the test phase, COLEMAN starts acting before the test execution. In this moment, the approach uses a MAB policy to prioritize a test case set available (T) from the current commit (c). Thus, the test cases of the prioritized test case set (T') are executed until the available test budget is reached. Feedback (reward) about T' applied is obtained and used by the MAB policy to adapt its experience for future actions (*online learning*). In the end, a report is generated, and the developers are informed.

COLEMAN can use different reward functions and MAB policies. In our previous experiments conducted [15], COLEMAN was evaluated using two reward functions: *Reward Based on Failures* (RNFail) and *Time-Ranked Reward* (TimeRank), and the *Fitness-Rate-Rank based on Multi-Armed Bandit* (FRRMAB) policy presented the best performance. Because of this, it is used in this study.

2.1 Reward Functions

Let t'_c to be a test case from a prioritized test case set T'_c at a commit (cycle) c . The first reward function *RNFail* (Equation 1) is based on the number of failures associated with a test case $t'_c \in T'_c$. This function captures the ability of a test case to produce failures.

$$RNFail(t'_c) = \begin{cases} 1 & \text{if } t'_c \text{ failed} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The second function is *TimeRank*, defined in Equation 2. Let T'^{fail} is composed by the failing test cases from T'_c ; The $prec(t'_{c_1}, t'_{c_2})$ function returns 1 if the position in T'_c of t'_{c_1} is lower than the position of t'_{c_2} .

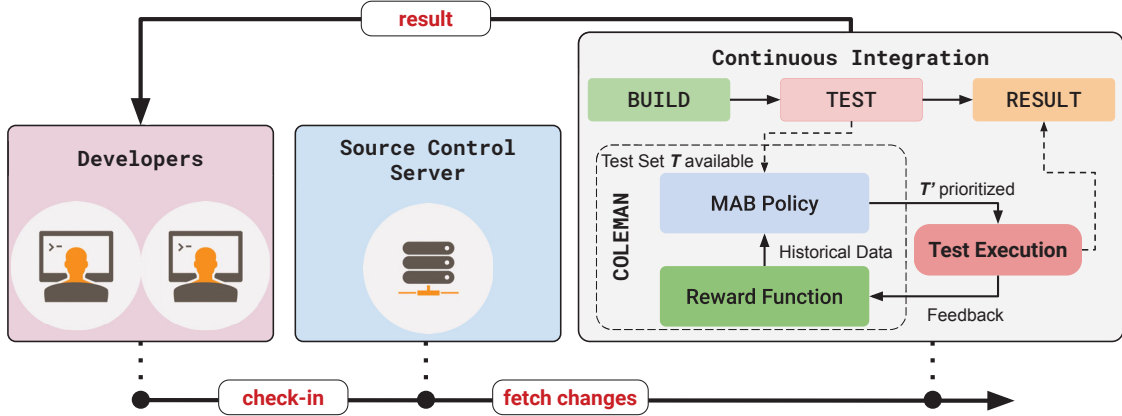


Figure 1: Overview of the COLEMAN interaction with the CI environment.

$$TimeRank(t'_c) = \frac{|T'^{fail}| - [\neg(fails(t'_c)) \times \sum_{i=1}^{|T'^{fail}|} prec(t'_c, t'_{c_i})]}{|T'^{fail}|} \quad (2)$$

The *TimeRank* function observes the rank of each t'_c in T'_c and considers the problem of early scheduling. This function privileges the failing test cases ranked in the first positions in T'_c , and it penalizes those that do not fail and precede failing ones. A non-failed test case receives a reward given by the accumulated number of test cases that failed until its position in the prioritization rank. That is, it receives a reward decreased by the number of failing test cases ranked after it.

2.2 FRRMAB

FRRMAB is a state policy that works with a sliding window SW as a smoother way to consider dynamic environments, allowing the observation of the changes in the quality of the arms (test cases) along the search process. Such a characteristic fits with the particularities inherent of the TCPCI problem. In such a scenario, COLEMAN treats the *arm* as a *test case* and the *turn* as a *commit*. In FRRMAB policy, the best test case t is chosen from T to compose the prioritized test case set T' in a commit c according to Equation 3:

$$\text{Select } t_c = \underset{t \in T}{\operatorname{argmax}} \left(FRR_{t,c} \quad C \times \sqrt{\frac{2 \times \ln \sum_{j=1}^T n_{j,c}}{n_{t,c}}} \right) \quad (3)$$

where the goal is, at each commit (c), to select the best test case from the test case set available which has an empirically estimated value ($FRR_{t,c}$) in a range that depends on the number of times ($n_{t,c}$) that has been applied previously. The parameter C controls the trade-off between exploitation and exploration.

Equation 3 considers the traditional behavior of a MAB policy, in which only a test case t is chosen from T available in a commit. To deal with the TCPCI scenario, Equation 3 could be performed multiple times, and in each time, a test case would be moved from T to compose a prioritized test case set T' . Thus, the process would repeat until no more test cases are available in T , and the order

of choice would define the ranking in T' . However, this can be costly [15]. Thus, FRRMAB policy was modified to evaluate all test cases and order them. In this way, the best test case is put at the top, followed by the second best one, and so on. A tie is broken at random. For the case when a new test case appears, its values are assigned with zero for the estimated value, and the number of times it has been applied previously.

FRRMAB uses a credit assignment procedure (Algorithm 1) to compute the *FRR* value for each test case. This procedure updates the test case values after the feedback from T' is received. In this way, it takes into account the impact observed in the most recent applications, allowing a quick prioritization for the next commit.

Algorithm 1: Credit Assignment procedure from FRRMAB for TCPCI (adapted from [8]).

```

begin
   $T'_c \leftarrow$  Prioritized Test Case Set applied in the last commit  $c$ ;
  foreach  $t' \in T'_c$  do
     $Reward_{t'} = 0.0$ ;
  end
  end
  foreach element  $\in$  SlidingWindow do
     $i = \text{element.GetTestCase}()$ ;
     $FIR = \text{element.GetFIR}()$ ;
     $Reward_{t'} = Reward_{t'} \quad FIR$ ;
  end
  end
  Ranking the rewards in a descending order ( $Reward_{t'}$ );
   $Rank_{t'} = \text{ranking value of } Reward_{t'}$ ;
  foreach  $t' \in T'$  do
     $Decay_{t'} = D^{Rank_{t'}} \times Reward_{t'}$ ;
  end
  end
   $DecaySum = \sum_{i=1}^{|T'|} Decay_{t'}$ ;
  foreach  $t' \in T'$  do
     $FRR_{t'} = \frac{Decay_{t'}}{DecaySum}$ ;
  end
  end
end

```

To compute the *FRR* value for each test case, first, it is applied a rank-based method that uses the Fitness Improvement Rate (FIR) method to not use the raw value that could deteriorate the efficiency of the algorithm [8]. To obtain the FIR value, we used the value obtained through a reward function (see Section 2.1).

After, the FIR values are stored in a given Sliding Window (*SW*) organized as a first-in, first-out (FIFO) queue that is used to evaluate the *W* recent applications. In this way, most recent values are added to the end, while the oldest records are removed from the beginning to maintain a constant size.

The use of a *SW* allows evaluating a test case without it being hampered by its performance at a very early stage, which may be irrelevant to its current performance. Thus, it is guaranteed that the FIR information in *SW* refers to a current search (prioritization) situation [8]. Subsequently, the $Reward_t$ of a test case *t* is calculated by the sum of all FIR values for the test case *t* in *SW*. Next, a descending ranking of all rewards from the test case in *SW* is determined. Thus, it is defined as a $Rank_t$ that represents the ranking value of a test case *t*, which prioritizes the best test cases. Then, a decay factor $D \in [0, 1]$ is used in order to transform the initial reward according to the relative position about the reward from the other test cases ($Decay_t$). The lower the values of *D* the greater the influence of the best test case. Finally, the decayed values of the rewards are normalized, resulting in the *Fitness-Rate-Rank* (*FRR*) for each test case *t*.

3 FINDING NEAR-OPTIMAL SOLUTIONS

In our work, the near-optimal solutions (reasonable solutions to serve as a baseline), are generated by a Genetic Algorithm (GA). This section describes the implementation of such an algorithm.

3.1 Population Representation

Each individual in the population represents a possible prioritization (T') for the test case set available (*T*) in a commit (*c*). To represent an individual in the population we use an integer encoding, where a gene represents a test case ($t \in T$) according to Figure 2.

t_i	t_i	1	t_i	2	t_i	3	...	t_n
2	3	1	4	...	n			

Figure 2: Individual Representation.

The value from *i*-th gene represents the test case priority (the execution order) in the prioritization test case set T' , with integer numbers in a range between $[0, |T| - 1]$. Thus, the GA is designed as a permutation problem.

3.2 Fitness Function

To compose the fitness function, we use the Normalized Average Percentage of Fault Detected (NAPFD) [18] (Equation 4) metric. This is a common metric used to evaluate a prioritized test case set under time constraints concerning failure detection effectiveness when not all of them are executed [17, 20]. NAPFD values range

from zero to one, in which higher values indicate how fast the faults are detected using fewer test cases.

$$NAPFD(T'_t) = p - \frac{\sum_1^n rank(T'_t)}{m \times n} \quad \frac{p}{2n} \quad (4)$$

where *m* is the number of faults detected by all test cases; $rank(T'_t)$ is the position of T'_t in T' , if T'_t did not reveal a fault we set $T'_t = 0$; *n* is the number of tests cases in T' ; and *p* is the number of faults detected by T' divided by *m*. As we desire the best prioritization, the problem is treated as a maximization problem.

3.3 Algorithm

Algorithm 2 illustrates the proposed GA. Similarly to COLEMAN, GA acts in each commit (*c*). Based on the test case set available in each commit (*T*), the execution time of the overall test case set is obtained (A_c), and a time budget is defined (TB_c). It is essential to highlight that the test results are already *known*. In this way, the original time to run the test is available. Besides, in order to observe the influence of time constraints, inherent to TCPPI problem, and how it affects the learning process, we defined three time budgets: 10%, 50%, and 80%.

Algorithm 2: Genetic Algorithm to Find Near-Optimal Solutions for TCPPI.

```

forall commit c in System do
     $T_c \leftarrow$  Test Case set available from system in the current
    commit;
     $A_c \leftarrow$  Available (total) time spent to run the tests;
     $TB_c \leftarrow$  Time Budget (10%, 50%, or 80% from  $A_c$ );
     $P_n \leftarrow$  Population initialization;
    Evaluate( $Q_n, TB_c$ );
    while stop criteria is not achieved do
         $Q_n \leftarrow$  Selection( $P_n$ );
         $Q_n \leftarrow$  Crossover( $Q_n$ );
         $Q_n \leftarrow$  Mutation( $Q_n$ );
        Evaluate( $Q_n, TB_c$ );
        Replacement( $Q_n, P_n$ );
    end
     $NAPFD_c \leftarrow$  Value obtained by best solution from  $P_n$ ;
end

```

In the algorithm, the NAPFD metric is used as a fitness function to guide our GA in the search space. At the moment of the optimization, an initial population is generated where each individual (solution) represents a possible prioritization order. Such a population is evaluated considering the time budget available. After, the optimization is performed, where the algorithm seeks to find the best prioritization. The optimization is executed until a stop criterion is achieved. Finally, the NAPFD value from the current commit is obtained from the best solution (with the highest NAPFD value).

3.4 Implementation Aspects

The algorithm was implemented using the DEAP framework [2] and is available in our the replication package [16]. We used the

operators available in the framework that fits with our population representation. They are:

- **Selection:** *Tournament Selection*. Picks randomly individuals from the population and selects the fittest one;
- **Crossover:** *Partially Matched Crossover* [4]. Two individuals are modified in place, and two random cut points are defined in each individual. One parent’s string is mapped into the other parent’s string and the remaining information is exchanged;
- **Mutation:** *Shuffle Indexes Mutation*. Shuffles the attributes from an individual according to a probability of each attribute to be moved;

4 EXPERIMENTAL SETUP

According to our goal, the experiment was guided by the following general research question: **RQ**: “How are the solutions generated by COLEMAN compared to the near-optimal solutions found by a GA?”. To answer this question, we performed a trade-off analysis of the solutions generated by COLEMAN, and reported in [15], regarding early fault detection and prioritization time. Such an analysis had as the baseline the solutions obtained in 30 independent executions of the GA algorithm.

4.1 Evaluation measures

In this study, we used three measures. The first one is the NAPFD (Section 3.2) which better fits the CI time constraints (budgets) [20]. The second metric is *Root-Mean-Square-Error* (RMSE), which is used to observe the difference between the predicted and the observed values. In this study, we use RMSE (Equation 5) to compute the differences between the NAPFD values in each CI Cycle (commit) c , obtained by COLEMAN (\hat{s}_c) and the near-optimal value T' (s_c) found by the GA. For an algorithm \mathcal{A} , the RMSE is computed as follows:

$$RMSE(\mathcal{A}) = \sqrt{\frac{\sum_{c=1}^{CI} (\hat{s}_c - s_c)^2}{CI}} \quad (5)$$

where CI is the amount of CI cycles in a system. Smaller RMSE values mean more accurate algorithms. Due to the hard task to find an optimal prioritization, s_c found by GA represents an approximation of the optimal prioritization.

The last metric used is the Prioritization Time. This metric takes into account the runtime, in seconds, to perform the prioritization.

4.2 Statistical Analysis

We applied statistical tests and effect size measure to evaluate the results. To determine the significance level, we used Kruskal-Wallis [6] and Mann-Whitney [10] statistical tests with a confidence level of 95%. On the other hand, to calculate the effect size magnitude (*Negligible*, *Small*, *Medium*, and *Large*) of the difference between two groups, we used the Vargha and Delaney’s \hat{A}_{12} [21] metric.

4.3 Target Systems

In order to provide a fair comparison and to maintain the reproducibility of the research, we used the same systems used in our previous work [15]. The seven target systems are detailed in Table 1. The first column shows the system name, followed by the number of

builds identified, and in parenthesis the number of builds included in the analysis. The third contains the total of failures found, and in parenthesis the number of builds in which at least one test failed. The last column shows the number of different (unique) test cases identified from build logs, and in parenthesis the range of test cases executed in the builds.

Table 1: Description of the Target Systems

Name	Builds	Failures	Test Cases
Druid	286 (168)	270 (71)	2391 (1778-1910)
Deeplearning4j	3410 (483)	777 (323)	117 (1-52)
Guava	2011 (1689)	7659 (112)	568 (308-512)
IOF/ROL	2392 (2392)	9289 (1627)	1941 (1-707)
OkHttp	9919 (6215)	9586 (1408)	289 (2-75)
Retrofit	3719 (2711)	611 (125)	206 (5-75)
ZXing	961 (605)	68 (11)	124 (81-123)
Total	22125 (14263)	28260 (3677)	5636 (1-1910)

4.4 Execution Parameters

We used the results obtained by COLEMAN available in [15]. In such work, the parameters are: sliding window size SW equals to 100, the coefficient C to balance exploration and exploitation equals to 0.3, and decayed factor equals to 1. Besides, in this work we evaluated both reward functions (see Section 2.1).

The parameters for the GA were defined empirically. The population size is 100 individuals, 80% of crossover probability, 10% of mutation probability, and stopping criteria of 100 generations (corresponding to 10,000 fitness evaluations). All the experiments were performed on an Intel® Xeon® E5-2450 with 2.10 GHz CPU, 47GB RAM, running Linux Ubuntu 18.04.1 LTS.

The results from GA and COLEMAN were obtained from 30 independent executions for each algorithm, considering three time budgets: 10%, 50%, and 80% of the overall execution time test case set available in each commit similarly to previous work [15].

4.5 Threats to Validity

In this section, we identify possible threats to the validity of our results, according to the taxonomy of Wohlin et al. [22].

Internal Validity: the parameter setting can be considered a threat. To minimize this threat, we used an empirical configuration for GA and, for COLEMAN, we adopted parameters used in related work. It is possible that using an automatic configuration setting the results could improve.

External Validity: we used only seven systems. Thus, the results cannot be generalized. However, our study provides some evidences towards an initial validation. New experiments should be performed. Besides that, we believe that our study can be easily replicated, using the raw data analyzed and disseminated in our replication package.

Conclusion Validity: the randomness of the algorithms is a threat. To minimize this threat, we executed the GA 30 times. Another threat is related to the statistical tests used. To minimize this threat, we used tests commonly adopted for non-deterministic algorithms in software engineering problems.

The last threats are concerning the fitness used to guide the GA in the search space, and the RMSE magnitude scale. On one hand, NAPFD was chosen because it better fits in the CI environment with time budgets. Our results may be different for other functions. On the other hand, the RMSE magnitudes were obtained based on our analysis, observations, SUTs behavior, and correlating the NAPFD and RMSE values. Other researchers can observe different aspects and propose a different scale.

5 RESULTS

In this section, the experimental results are presented and analyzed, aiming at answering the posed question.

In order to answer the research question, we computed the NAPFD, RMSE, and Prioritization Time averages for GA and COLEMAN using both reward functions *RNFail* and *TimeRank*. Table 2 shows the obtained mean values \pm standard deviation. The average was computed from 30 independent executions in each system and time budget. Values highlighted in bold are the best. Besides, for the NAPFD measure, values that are statistically equivalent to the best ones have their corresponding cells painted in light gray. Different symbols represent the effect size magnitudes (see caption of the table). In each time budget, for each comparison in a system, the Kruskal-Wallis test was performed.

Table 2 shows for all systems a relevant difference between GA and COLEMAN execution time. As expected, COLEMAN is much faster than GA. An expressive difference is observed in the *Druid* system, a case that shows how hard the prioritization can be, even for our GA that knows a priori the final results. However, how does such a gain in time impact in the NAPFD values? Does more time imply better prioritizations? Next, we investigate such a trade-off, by analyzing NAPFD and RMSE.

Regarding the NAPFD values, GA outperforms COLEMAN in all systems and all time budgets with a *large* magnitude and statistical difference. This is expected, as mentioned before, the GA previously *knows* the testing result and is required only to find an approximation from the optimal prioritization. In this way, the results found by GA are used as a baseline to observe how distant the prioritization proposed by COLEMAN is.

Considering this aspect, in some cases, the NAPFD values reported by GA are close to the ones obtained by COLEMAN. They suggest COLEMAN proposes near-optimal solutions. To a better visualization, Figure 3 illustrates radar charts (or spider graphs) for each time budget. Each angle represents the NAPFD value for a system. The green line represents the values found by GA, orange line by *TimeRank*, and blue by *RNFail*.

The figure shows that increasing the time budget (less time constraint), COLEMAN produces solutions closest to optimal, in which *TimeRank* performs a bit better than *RNFail*. Besides, in some systems, for instance, in the *Deeplearning4j* system, we observe that even increasing the time budget, this difference continues the same between the values found by both reward functions and by GA. In order to understand this behavior, we generated Figure 4.

The figure presents, in overall (in the same scale), different information about each system: number of valid builds, number of

failures, number of failed builds, number of test cases, mean number of failures by cycles, and mean number of failing cycles. More information is found in Table 1.

As we can observe, *Deeplearning4j* has a high average of failing builds. However, only this does not help COLEMAN to provide a good prioritization. Because of this, we analyzed other relevant characteristics that may impact the prioritization, such as the test case volatility. In such a system, we observed a high test case volatility¹, mainly in the failing test cases, consequently, hindering the prioritization. This explains the difference across time budgets. Such behavior also impacts in the GA performance, which is the second worst for such a system in terms of prioritization time.

The *ZXing* system is the simplest one. Thus, such information support to understand the closest values in the *ZXing* system between COLEMAN and GA. In this system, there is low test case volatility, and there are peaks in the failure detection in a few commits with long periods without failures. This scenario endorses an approach based on historical test data.

The highest difference between COLEMAN and GA can be observed in the *Druid* system in the presence of the most restrictive time budget (10%). This system has some particularities: (i) although it has a high average of failing cycles, what would benefit a failure history approach, it has two peaks of failures which increases the average of failing cycles; (ii) a large test case set, in which many failures are distributed in many test cases; and (iii) few number of CI Cycles. This hampers the failure-history prioritization performed by COLEMAN in a very restrictive scenario. Another reason for the bad performance from COLEMAN can be related to the sliding window size used (100 cycles). A sliding window with a small size can improve the results for systems with few numbers of CI cycles (commits).

Nevertheless, the difficulty in finding good prioritizations in the *Druid* system is not exclusive of COLEMAN. The particularities from this system also negatively affect the GA, which required a lot of time to find the near-optimal solutions for this system.

The worst NAPFD values are obtained in the *IOF/ROL* system for both algorithms. In this system, the values for both GA and COLEMAN are far from the maximum value for the metric, which is 1. As we can observe in Figure 4, there is a considerable amount of information to be used in the learning process. To a more in-depth understanding, we analyzed the system behavior over the CI Cycles according to the failures and test case volatility. This system encompasses a high test case volatility coupled with a great number of failures distributed over many test cases. The high test case volatility may be associated with a pre-submit strategy [3], because the test cases reappear in some cycles. Consequently, this hampers to find reasonable solutions for the TCPCPI problem.

The NAPFD metric gives us a quantitative notion about prioritization quality, and the radar charts a better visualization for comparison. However, to effectively calculate how far the COLEMAN solutions are from the GA near-optimal solutions, we use the RMSE values, presented in Table 2. In such a table, low values represent near-optimal solutions. We can observe *TimeRank* has the best values, in most cases with statistical difference. *RNFail* outperforms *TimeRank* in only one case, in the *Guava* system and time budget of 50%. To a better visualization, we generated Figure 5.

¹See supplementary material about characteristics of the systems.

Table 2: Values for the Evaluation Measures Obtained with GA and COLEMAN using both Reward Functions.

SUT	NAPFD			RMSE		PRIORITIZATION TIME (SEC.)		
	RNFail	TimeRank	GA	RNFail	TimeRank	RNFail	TimeRank	GA
TIME BUDGET: 10%								
Druid	0.6801 ± 0.052 ▲	0.7137 ± 0.074 ▲	0.9951 ± 0.000 ★	0.5173 ± 0.0793 ▲	0.4729 ± 0.1165 ▲	0.2383 ± 0.044	0.2316 ± 0.043	31.5148 ± 34.954
Deeplearning4j	0.7533 ± 0.002 ▲	0.7716 ± 0.000 ▲	0.8137 ± 0.000 ★	0.1512 ± 0.0031 ▼	0.1076 ± 0.0008 ★	0.0272 ± 0.002	0.0272 ± 0.002	1.8732 ± 1.254
Guava	0.9554 ± 0.002 ▲	0.9586 ± 0.001 ▲	0.9976 ± 0.000 ★	0.1973 ± 0.0030 ▼	0.1923 ± 0.0023 ▼	0.0660 ± 0.019	0.0660 ± 0.019	1.6643 ± 4.228
IOF/ROL	0.3632 ± 0.001 ▲	0.3670 ± 0.001 ▲	0.4081 ± 0.000 ★	0.1796 ± 0.0026 ▼	0.1704 ± 0.0030 ▼	0.0277 ± 0.004	0.0277 ± 0.004	0.9125 ± 1.693
OkHttp	0.8323 ± 0.000 ▲	0.8407 ± 0.000 ▲	0.8886 ± 0.000 ★	0.2200 ± 0.0000 ▼	0.2064 ± 0.0000 ▼	0.0286 ± 0.002	0.0285 ± 0.002	0.6554 ± 0.983
Retrofit	0.9639 ± 0.000 ▲	0.9642 ± 0.000 ▲	0.9712 ± 0.000 ★	0.0802 ± 0.0000 ★	0.0790 ± 0.0000 ★	0.0277 ± 0.002	0.0277 ± 0.002	0.2796 ± 0.517
ZXing	0.9826 ± 0.000 ▲	0.9828 ± 0.000 ▲	0.9998 ± 0.000 ★	0.1280 ± 0.0001 ★	0.1273 ± 0.0001 ★	0.0343 ± 0.002	0.0342 ± 0.002	0.4474 ± 0.874
TIME BUDGET: 50%								
Druid	0.9333 ± 0.013 ▲	0.9710 ± 0.008 ▲	0.9941 ± 0.000 ★	0.1689 ± 0.0366 ▼	0.1087 ± 0.0329 ★	0.2474 ± 0.040	0.2373 ± 0.041	31.4262 ± 34.849
Deeplearning4j	0.7890 ± 0.001 ▲	0.8200 ± 0.000 ▲	0.9024 ± 0.000 ★	0.2423 ± 0.0014 ▼	0.1742 ± 0.0006 ▼	0.0271 ± 0.002	0.0271 ± 0.002	1.8770 ± 1.255
Guava	0.9653 ± 0.004 ▲	0.9675 ± 0.007 ▲	0.9994 ± 0.000 ★	0.1693 ± 0.0083 ▼	0.1708 ± 0.0169 ▼	0.0648 ± 0.018	0.0648 ± 0.018	1.6790 ± 4.255
IOF/ROL	0.5046 ± 0.002 ▲	0.5189 ± 0.002 ▲	0.5790 ± 0.000 ★	0.2130 ± 0.0042 ▼	0.1906 ± 0.0040 ▼	0.0278 ± 0.005	0.0278 ± 0.005	0.9185 ± 1.695
OkHttp	0.9192 ± 0.000 ▲	0.9317 ± 0.000 ▲	0.9544 ± 0.000 ★	0.1536 ± 0.0004 ▼	0.1277 ± 0.0006 ★	0.0283 ± 0.002	0.0283 ± 0.002	0.6507 ± 0.983
Retrofit	0.9853 ± 0.000 ▲	0.9893 ± 0.000 ▲	0.9946 ± 0.000 ★	0.0836 ± 0.0000 ★	0.0644 ± 0.0000 ★	0.0277 ± 0.002	0.0277 ± 0.002	0.2867 ± 0.519
ZXing	0.9846 ± 0.000 ▲	0.9857 ± 0.000 ▲	0.9997 ± 0.000 ★	0.1153 ± 0.0002 ★	0.1110 ± 0.0002 ★	0.0343 ± 0.003	0.0343 ± 0.003	0.4507 ± 0.874
TIME BUDGET: 80%								
Druid	0.9380 ± 0.012 ▲	0.9830 ± 0.003 ▲	0.9942 ± 0.000 ★	0.1484 ± 0.0325 ★	0.0640 ± 0.0180 ★	0.2518 ± 0.039	0.2393 ± 0.041	31.5536 ± 34.989
Deeplearning4j	0.8424 ± 0.001 ▲	0.8641 ± 0.001 ▲	0.9519 ± 0.000 ★	0.2110 ± 0.0031 ▼	0.1804 ± 0.0013 ▼	0.0272 ± 0.002	0.0272 ± 0.002	1.8745 ± 1.254
Guava	0.9784 ± 0.012 ▲	0.9841 ± 0.014 ▲	0.9995 ± 0.000 ★	0.1120 ± 0.0500 ★	0.0870 ± 0.0567 ★	0.0655 ± 0.018	0.0649 ± 0.018	1.6609 ± 4.261
IOF/ROL	0.5569 ± 0.002 ▲	0.5678 ± 0.001 ▲	0.6096 ± 0.000 ★	0.1709 ± 0.0040 ▼	0.1568 ± 0.0034 ★	0.0279 ± 0.005	0.0279 ± 0.005	0.9205 ± 1.694
OkHttp	0.9350 ± 0.000 ▲	0.9478 ± 0.000 ▲	0.9606 ± 0.000 ★	0.1130 ± 0.0004 ★	0.0794 ± 0.0007 ★	0.0284 ± 0.002	0.0284 ± 0.002	0.6526 ± 0.983
Retrofit	0.9881 ± 0.000 ▲	0.9916 ± 0.000 ▲	0.9972 ± 0.000 ★	0.0808 ± 0.0000 ★	0.0652 ± 0.0000 ★	0.0277 ± 0.002	0.0277 ± 0.002	0.2927 ± 0.519
ZXing	0.9972 ± 0.000 ▲	0.9996 ± 0.000 ▲	0.9997 ± 0.000 ★	0.0271 ± 0.0005 ★	0.0014 ± 0.0005 ★	0.0343 ± 0.002	0.0343 ± 0.002	0.4531 ± 0.872

This table reports NAPFD and Prioritization Time (averages ± standard deviation), and RMSE values obtained from 30 independent runs and organized by each time budget evaluated (10%, 50%, and 80% of the available time). Values highlighted in bold denote the best algorithm. Results from the statistical test applied using NAPFD values are also presented: “★” denotes the best algorithm for a time budget in a SUT. Results in gray are that statistically equal to the best one; “▼” indicates that negligible effect size; “▼” denotes a small magnitude, “▲” a medium magnitude, and “▲” large. (see Section 4.1 for more information). Regarding to RMSE values, different symbols are used to represent the distance from optimal prioritization (see Eq. 6): a “★” symbol which denotes the *very near* category; “▼” indicates the category *near*, “▼” denotes *reasonable*, “▲” *far*, and “▲” *very far*.

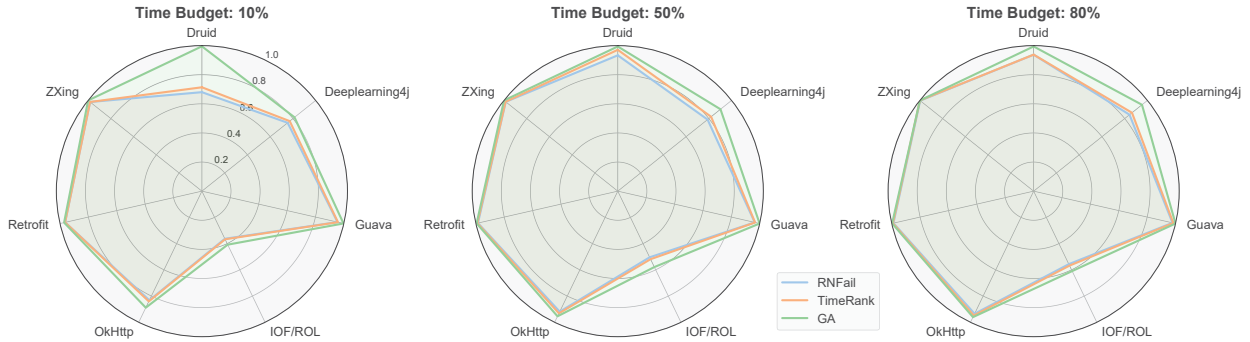


Figure 3: Radar charts from NAPFD values found by GA, TimeRank, and RNFail.

As we can observe, both reward functions have similar behavior, but the significant difference between them appears in the time budget of 50%. This time budget presents a constraint that allows better comparison while keeping the difficulty inherent to the problem [20]. Besides, increasing the time budget, smaller RMSE values are obtained, even near-optimal values, as reported in the ZXing system. We observe in the figures that *TimeRank* is better in the hard cases, Deeplearning and Druid.

This metric is adequate to evaluate an approach effectively. For instance, both algorithms reported small NAPFD values for the

IOF/ROL system, but this occurs due to its characteristics, the challenges inherent in finding reasonable solutions. In this system, all RMSE values are around 0.2, which represents solutions close to the optimal.

The worst RMSE values are for the system Druid, in the time budget of 10%, with $RMSE \approx 0.5$. As mentioned before, the bad performance is associated with the characteristics of the system and with the consistency of the historical data.

In order to provide a better representation of the RMSE values, we defined an RMSE scale of magnitude. Such a scale shows the distance from near-optimal solutions, as follows:

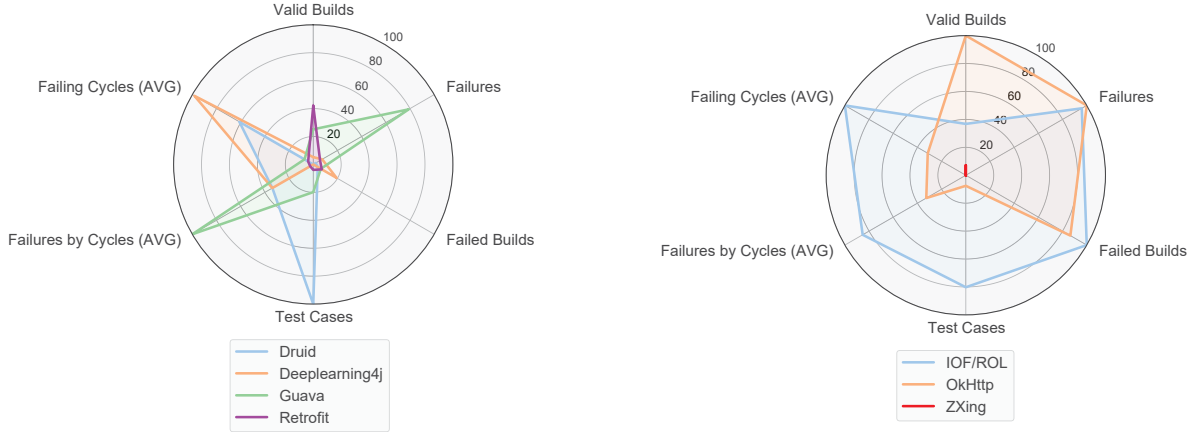


Figure 4: Radar charts about the systems characteristics.

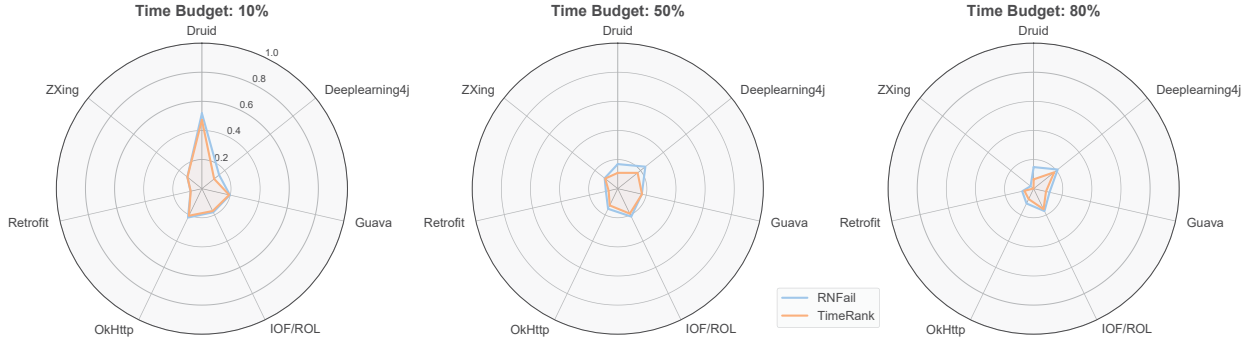


Figure 5: Radar charts from RMSE values from *TimeRank* and *RNFail*.

$$\text{RMSE Magnitude} = \begin{cases} \text{very near} & \text{if } \text{RMSE} < 0.15 \\ \text{near} & \text{if } 0.15 \leq \text{RMSE} < 0.23 \\ \text{reasonable} & \text{if } 0.23 \leq \text{RMSE} < 0.30 \\ \text{far} & \text{if } 0.30 \leq \text{RMSE} < 0.35 \\ \text{very far} & \text{if } 0.35 \leq \text{RMSE} \end{cases} \quad (6)$$

where the *very near* category represents an approximated optimal performance. The *near* category represents solutions that are reaching optimal performance, and some improvements are necessary. The *reasonable* category represents the minimum acceptable performance. The solutions are acceptable. In such a category, the target system behavior, or possibly the constraints, can make it hard for the approach to obtain better solutions. The *far* category means the solutions provided are not satisfactory. Approach performance requires meaningful improvements. Finally, the *very far* category represents the solutions that are far away from to be useful and considered reasonable.

Based on the RMSE scale of magnitude, we identified that *TimeRank* obtains *very near* optimal solutions in 13 cases (out of 21 - 62%, 7 systems \times 3 time budgets) and 7 cases (\approx 33%) in the category *near*, whilst *RNFail*, respectively, in 8 (38%) and 10 (\approx 48%) cases. Besides,

RNFail has 1 case in the *reasonable* category. Both reward functions have one case in the *very far* category, in the Druid system for the time budget of 10%. On the other hand, in both reward functions the better solutions are obtained for the time budget of 80%, what is aligned with previous analyses.

Considering the analysis reported in this section, we can conclude that the prioritization time had no significant negative impact in the COLEMAN solutions. The time spent by COLEMAN to prioritize is negligible. This metric also helped to observe the difficulty inherent of the TCPCI problem though the values reported by GA. This algorithm, in some systems, spent a lot of time to prioritize. The difference in the prioritization time values from GA and COLEMAN shows how COLEMAN mitigates adequately the TCPCI problem reaching solutions near to the GA solutions in most times.

Answering our RQ: COLEMAN provides solutions that are near (or very near) to the GA solutions in 38 cases (out of 42, \approx 90%) considering all systems, budgets and both reward functions. It spends a negligible time (less than one second) even in challenging systems for the TCPCI problem.

5.1 Discussion

The last section presented results of an in-deeper analysis, performed to understand the relevant characteristics that impact the COLEMAN. In this section, we discuss the main findings highlighting some research directions to COLEMAN and learning TCPCI approaches.

We observe that the COLEMAN performance is better in the presence of a less restrictive budget, that is, in this scenario, the solutions are very near to the optimal. As COLEMAN is based on the failure-history, a greater quantity of cycles and few peaks of failures seems to make the prioritization easy. On the other hand, we observe some factors that hamper the prioritization: high volatility of failed test cases, a large test case set, in which many failures are distributed in many test cases, and a small number of CI Cycles make the learning hard, mainly in the presence of restrictive budgets.

Few cycles may imply a small failure history and low performance. A way to solve this problem is to reduce the sliding window size used by COLEMAN. This may get better results. Another research direction to be investigated is the use of a hybrid approach, to mitigate the limitation of starting without learning.

TimeRank performs a bit better than *RNFail*. A great difference is observed in the less restrictive budget. It seems that the use of other information besides the ability of a test case to produce failures can improve the performance. Maybe other reward functions should be explored, such as one that considers the test case execution time. Other factors that are costly to calculate such as coverage should be avoided.

6 RELATED WORK

TCPCI approaches have been proposed recently [17]. We can mention the works of Marijan et al [13]. The first approach these authors introduced is named ROCKET that considers the distance of the failure status from a current execution of a test case to its execution time. But such an approach does not consider the total history of failures. A posterior work [11] considers, given a test budget, other fault detection, business, performance, and technical perspectives. But these works need additional information related to coverage and features. The tool, called TITAN, also proposed by Marijan et al [14], uses constraint programming to minimize the number of test cases that cover some requirements that the original test cases cover. Then, the minimized set is prioritized using ROCKET [13]. The problem with these works is the cost, because feature coverage or other additional information are required. In the most recent work, they propose a learning algorithm to classify test cases considering the feature coverage. The main idea is to discover redundant test cases in the context of Highly Configuration Systems (HCS), taking into account the test budget. The priority is calculated based on its historical fault detection effectiveness. But the approach also required an effort to calculate the coverage.

The work of Xiao et al. [23] focuses on test cases that failed recently, to determine a priority for the test cases in a same commit and after, orders them considering the failure history, test coverage, test size and execution time. Haghighatkah et al. [5] tries to improve the effectiveness of the prioritization by using historical failure knowledge with a diversity measure, calculated by comparing the text from test cases. The idea is not to calculate similarity

based on measures that rely on code such as coverage, call methods and so on. Busjaeger and Xie [1] use (SVM^{map}) to create a prediction model for the fault-proneness of test cases to be used in the prioritization, taking into account five attributes: test coverage of modified code, textual similarity between tests and changes, recent test-failure or fault history, and test age. But these attributes need additional information and rely on code instrumentation.

The works mentioned above present some limitations. They require code analysis, what can be costly. They do not address the main characteristics of CI environments. For instance, they do not consider test case volatility. They are not adaptive, that is, they do not learn with past prioritizations. To overcome such limitations, learning based approaches were proposed. They are: COLEMAN [15], described in Section 2), and RETECS [20]. This last one is an approach based on Reinforcement Learning. It takes as input the test case duration, historical failure data, and previous last execution and guided by a reward function that learns over the cycles and can adapt to changes. RETECS presents performance, regarding NAPFD (*Normalized Average Percentage of Faults Detected*), comparable with deterministic methods [20]. However, COLEMAN outperformed the RETECS, and can be considered as the state-of-the-art in TCPCI [15]. This is the reason COLEMAN was used in our analysis to evaluate the current status of TCPCI.

We can see analyzing the above-mentioned works that search-based approaches are not proposed for the TCPCI. They are complex and take much time to execute, although they are capable of performing an optimization and produce near-optimal solutions. However, we have not found a study similar to ours that uses them as a baseline to analyze the trade-offs, regarding early-fault detection and prioritization time. Our study is the first one evaluating the gains and losses of the learning approaches.

7 CONCLUDING REMARKS

This paper presented results from a trade-off analysis of the solutions produced by COLEMAN, a MAB-based approach for the TCPCI problem, having as baseline near-optimal solutions found by a GA. We evaluated COLEMAN using the FRRMAB policy and two reward functions: Reward Based on Failures and Reward Based on Time-Rank. The analysis was conducted using seven large-scale real-world software systems, three different time budgets, and three measures: NAPFD, representing the prioritization quality according to early fault detection capability; RMSE, measuring the distance between COLEMAN and GA solutions; and Prioritization Time.

The evaluation shows that in most systems, except by one, COLEMAN yields near-optimal solutions with negligible time. The unique exception is under a restrictive test budget associated with a few historical data. Besides, we observed that a high test case volatility, mainly in failing test cases, associated with a high number of failures distributed over many test cases hampers an adequate prioritization. Such a difficulty makes the problem hard, even for GA, that took a lot of time to execute.

The results presented here can serve as guidance to evaluate future approaches in the TCPCI context. Furthermore, the characteristics of the systems can be used by the researchers to redirect their research to better results. In fact, we intend to use this paper in future work as a basis to derive and evaluate new approaches. In

this sense, future work includes the use of other evaluation measures as a fitness function to the GA. Besides, other systems with a greater number of failures and test cases should be used to evaluate scalability.

ACKNOWLEDGMENTS

The work is supported by the Brazilian funding agencies CAPES and CNPq (Grant 305968/2018).

REFERENCES

- [1] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 975–980.
- [2] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: A Python Framework for Evolutionary Algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12)*. Association for Computing Machinery, New York, NY, USA, 85–92.
- [3] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 235–245.
- [4] David E. Goldberg and Robert Lingle. 1985. AllelesLociand the Traveling Salesman Problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*. L. Erlbaum Associates Inc., USA, 154–159.
- [5] Alireza Haghighathkhan, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98.
- [6] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621.
- [7] Volodymyr Kuleshov and Doina Precup. 2014. Algorithms for multi-armed bandit problems. *Journal of Machine Learning Research* 1 (2014), 1–48.
- [8] K. Li, A. Fialho, S. Kwong, and Q. Zhang. 2014. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on* 18, 1 (2014), 114–130.
- [9] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (April 2007), 225–237.
- [10] Henry B. Mann and Donald R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.
- [11] Dusica Marijan. 2015. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*. IEEE Computer Society, Washington, DC, USA, 157–162.
- [12] Dusica Marijan, Arnaud Gotlieb, and Marius Liaaen. 2019. A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience* 49, 2 (2019), 192–213.
- [13] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *IEEE International Conference on Software Maintenance*. IEEE, 540–543.
- [14] Dusica Marijan, Marius Liaaen, Arnaud Gotlieb, Sagar Sen, and Carlos Ieva. 2017. TITAN: Test Suite Optimization for Highly Configurable Software. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 524–531.
- [15] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments. *IEEE Transactions on Software Engineering* (2020), 12.
- [16] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. Supplementary Material - Multi-Armed Bandit Test Case Prioritization in Continuous Integration Environments: A Trade-off Analysis. <https://doi.org/10.17605/OSF.IO/J67EB>
- [17] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. 2020. Test Case Prioritization in Continuous Integration Environments: A Systematic Mapping Study. *Information and Software Technology* (2020).
- [18] Xiao Qu, Myra B. Cohen, and Katherine M. Wolf. 2007. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *IEEE International Conference on Software Maintenance*. 255–264.
- [19] Herbert Robbins. 1985. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*. Springer, 169–177.
- [20] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 12–22.
- [21] Andras Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (jan 2000), 101–132.
- [22] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.
- [23] Lei Xiao, Huaikou Miao, and Ying Zhong. 2018. Test case prioritization and selection technique in continuous integration development environments: a case study. *International Journal of Engineering & Technology* 7, 2.28 (2018), 332–336.
- [24] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification & Reliability* 22, 2 (March 2012), 67–120.

APPENDIX D – AN EVALUATION OF RANKING-TO-LEARN APPROACHES FOR TCPCI

An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration

Jackson A. Prado Lima
Department of Informatics
Federal University of Paraná (UFPR)
Curitiba, Paraná, Brazil
japlima@inf.ufpr.br

Silvia R. Vergilio
Department of Informatics
Federal University of Paraná (UFPR)
Curitiba, Paraná, Brazil
silvia@inf.ufpr.br

ABSTRACT

Continuous Integration (CI) environments is a practice adopted by most organizations that allows frequent integration of software changes, making software evolution more rapid and cost-effective. Such environments require dynamic Test Case Prioritization (TCP) approaches that adapt better to the test budgets and frequent addition/removal of test cases. In this sense, Ranking-to-Learn approaches have been proposed and are more suitable for CI constraints. By observing past prioritizations and guided by reward functions, they learn the best prioritization for a given commit. In order to contribute for improvements and direct future research, this work evaluates how far the solutions produced by these approaches are from optimal solutions produced by a deterministic approach. To this end, we consider two approaches that can be considered the state-of-the-art: i) *RETECS*, which is based on Reinforcement Learning; and ii) *COLEMAN*, an approach based on Multi-Armed Bandit. The evaluation was conducted with twelve systems, three test budgets, two reward functions, and six measures concerning fault detection effectiveness, early fault detection, test time reduction in the CI cycles, prioritization time, and accuracy. Our findings have some implications for the approaches application and reward function choice. The approaches are applicable in real scenarios and produce solutions very close to the optimal ones, respectively, in 92% and 75% of the cases. Both approaches have some limitations to learn with few historical test data (a small number of CI Cycles) and deal with a large test case set, in which many failures are distributed over many test cases.

KEYWORDS

Test Case Prioritization, Continuous Integration environments, Ranking-to-Learn

1 INTRODUCTION

Continuous Integration (CI) is a common practice adopted by many organizations to make software evolution more cost-effective and reliable. In a CI scenario, the software is changed, built, and tested many times in a short period. This is usually costly because a test suite often includes thousands of test cases and requires several hours or even days to execute [9]. In such a scenario, the application of regression testing techniques is fundamental.

Regression testing techniques are classified into three main categories [31]: minimization, selection, and prioritization. Techniques based on Test Case Minimization (TCM) usually remove redundant test cases, minimizing the test set according to some criterion. Test

Case Selection (TCS) selects a subset of test cases, the most important ones to test the software. Test Case Prioritization (TCP) attempts to re-order a test suite to identify an “ideal” order of test cases that maximizes specific goals, such as early fault detection. TCP techniques are very popular in the industry and have some advantages because they do not discard any test case. Moreover, the test cases that have a higher probability of detecting a fault are executed first. This allows interrupting the test activity early and reducing costs and time between each CI cycle.

Existing TCP techniques use different kind of information to prioritize the test cases: historical failure data, test coverage, requirements, and system models [31]. However, most techniques need adaptations to be applied in CI environments. This is because in CI there are limited resources and constraints that can make the application of a technique infeasible, due to the time spent for the prioritization or code analysis. The application of TCP techniques that require extensive code analysis or coverage, such as search-based ones, is not always possible due to the test budget for a build. To deal with these constraints, approaches have been proposed recently [4, 9, 17–20, 30]. Most of them are history-based, that is, they consider test cases that failed in the past are more likely to fail in the future. This kind of TCP technique has been acknowledged as one of the most suitable for the CI environment characteristics [9].

Nevertheless, these approaches present some limitations. Some of them require code analysis, which can be costly. They do not consider test case volatility, a characteristic associated with the fact that test cases may be added and/or removed over the cycles. They are not adaptive, that is, they do not learn with past prioritizations. To overcome such limitations, learning approaches based on historical failure data have been proposed [21, 27]. These approaches observe previous cycles and learn with past prioritizations guided by a reward function (online learning). We can mention two approaches that can be considered the state-of-the-art in TCP in CI environments (TCPCI): i) *RETECS*[27] (*Reinforced Test Case Selection*), which uses Reinforcement Learning (RL) to perform the prioritization; and ii) *COLEMAN*[21] (*Combinatorial VOLatiLE Multi-Armed BANDit*), an approach that uses a Multi-Armed Bandit (MAB) policy.

These two learning-based approaches are the focus of our work. They deal properly with the test case volatility and CI constraints. In evaluations conducted by the authors, they present promising results regarding standard TCP metrics such as Normalized Average Percentage of Faults Detected (NAPFD) and Average Percentage of Faults Detected with cost consideration (APFDc). The time spent to perform the prioritization is suitable for the CI budget. However, there are many difficulties inherent to the TCPCI problem, and we do not get there yet. Sure there is room for improvement. With this

in mind, this paper aims to evaluate how far the solutions produced by both learning-based approaches are from optimal solutions. To answer this question, we evaluate the approaches having a deterministic approach as a baseline. The deterministic approach knows the test results a priori and can generate the optimal solutions regarding fault detection. We report results from the use of two reward functions: Reward Based on Failures and Reward Based on Time-Rank, concerning twelve large-scale real-world software systems in the presence of three different time constraints (budgets). Six measures are used in the evaluation concerning: fault detection effectiveness, early fault detection, test time reduction in the CI cycles, prioritization time, and accuracy, regarding the distance from the optimal solution.

In this way, we analyze the approaches solutions' trade-offs regarding fault detection and prioritization time. This allows the assessment of lightweight test case prioritization approaches in CI environments by evaluating how far their solutions are from optimal ones. As contributions, our findings have some implications we discuss in this work, for: i) application of the approaches: we present guidelines that include the choice of the reward function, cost consideration regarding test case duration, and characteristics of the systems and budgets; ii) identification of limitations and possible improvements: we analyze some aspects regarding the number of test cases, failures distribution over test cases and CI cycles. Such aspects are drawbacks for the learning approaches and constitute gaps for directing future research; and iii) benchmark construction: we identify hard prioritization cases. In addition to this, we make available a replication package containing supplementary material¹.

The paper is structured as follows. Section 2 contains background about CI environments, TCP works for CI, and related work. Section 3.1 details evaluated approaches. Section 4 describes how our evaluation was conducted: objectives, *Systems Under Test (SUT)*, evaluation measures and used parameters. Section 5 shows and analyses the results. Section 6 presents the main threats to the validity of our results. Section 7 contains our final remarks and discusses future work.

2 BACKGROUND AND RELATED WORK

Continuous Integration (CI) environments have been increasingly adopted in the industry. CI environments automate the process of building and testing software, and allow engineers to merge code that is under development or maintenance with the mainline codebase at frequent time intervals. In CI development, teams work continuously integrating code and make smaller code commits every day, usually monitored by a CI server. When a change occurs, the CI server clones this code, builds it, and runs the testing processes. When the entire process is finished, the CI server generates a report (feedback), and the developers are informed. Figure 1 illustrates this process.

2.1 TCP in CI environments

Given a test case set T , available for a build, the set PT of all possible permutations of T , and a function f that determines the performance of a given prioritization T' from PT to real numbers, the TCPCI problem aims at finding the best T' to achieve certain specific criteria measured by f . In CI, the determination of T' may be subject to a test budget that is the available time to execute the CI cycle.

Many TCP approaches exist in the literature [11, 31]. Among them, we can mention approaches that use evolutionary algorithms [1, 5, 8, 15]. However, such approaches usually are complex and take much time to execute. In addition to this, most of them require coverage and code changes analysis, and do not consider the CI particularities such as volatility of test cases, multiple test requests, constraints and test budget. Because of this, approaches specific for CI have been proposed recently [23].

Marijan et al [19] introduce an approach named ROCKETS. Such an approach implements domain specific heuristics that consider the distance of the failure status from a current execution of a test case to its execution time. However, ROCKETS does not consider the entire history of failures. An extension is proposed [17] to consider, given a test budget, other fault detection, business, performance, and technical perspectives. Such an extension needs additional information related to coverage and features.

The tool, called TITAN [20], uses constraint programming to minimize the number of test cases that cover some requirements that the original test cases cover. Then, the minimized set is prioritized using ROCKETS [19]. The problem with these works is the cost because feature coverage or other additional information is required.

In the context of Highly Configuration Systems, Marijan et al [18] use a based-tree learning algorithm to classify test cases according to the feature coverage into the following categories: unique, totally redundant, and partially redundant. The main idea is to eliminate test cases totally redundant. Partially redundant test cases can also be included in the set according to its priority and time budget. The priority is calculated based on its historical fault detection effectiveness. Nevertheless, an effort to calculate the coverage is also necessary.

The work of Xiao et al. [30] determines a priority for the test cases in the same commit and after, orders them considering the failure history, test coverage, test size, and execution time. The technique focuses only on test cases that failed recently, not exploring new test cases. Haghighatkhah et al. [9] show the use of historical failure knowledge is a strong predictor for TCP in CI environments, and that it is effective to catch regression faults earlier without requiring a large amount of historical data. In addition to this, the effectiveness can be improved by using such knowledge with a diversity measure, calculated by comparing the text from test cases. The idea is not to calculate similarity based on measures that rely on code such as coverage, call methods, and so on.

Busjaeger and Xie [4] use SVM^{map} to create a prediction model for the fault-proneness of test cases to be used in the prioritization, taking into account five attributes: test coverage of modified

¹Our supplementary material is available at https://osf.io/x96fk/?view_only=020b612cbdd84fa38d6a974743f9d823. After publication, we will use a DOI as a reference for the material.

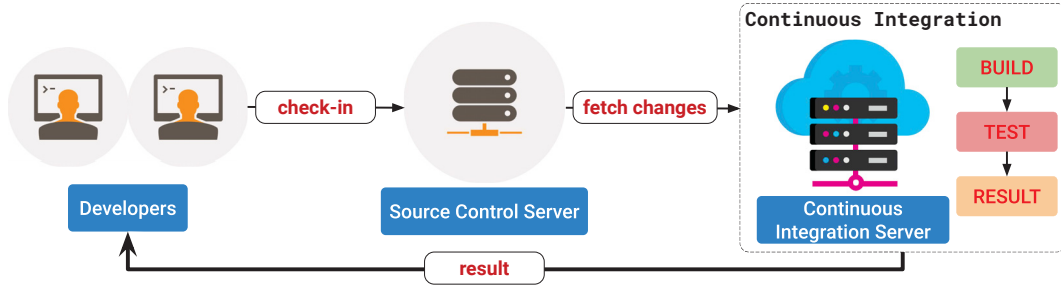


Figure 1: Overview of a Continuous Integration environment [21].

code, the textual similarity between tests and changes, recent test-failure or fault history, and test age. However, these attributes need additional information and rely on code instrumentation.

The works mentioned above present some limitations. They require code analysis, which can be costly. They do not address the main characteristics of CI environments. For instance, they do not consider test case volatility, a characteristic associated with the fact that test cases may be added and/or removed over the cycles. They are not adaptive, that is, they do not learn with past prioritizations.

To overcome such limitations, learning approaches based on historical failure data have been proposed. Bertolino et al [3] distinguish two kinds of TCP learning-based strategies. The first one, named Learning-to-Rank, uses supervised learning to train a model based on some test features. The model is then used to rank test sets in future commits. The problem with these strategies is that the model may no longer be representative when the commit context changes. The second kind, named Ranking-to-Learn, is more suitable to the dynamic CI context. This strategy learns based on the rewards obtained from the feedback of previously used ranks. The main idea is to maximize the rewards.

The work of Bertolino et al. [3] presents results comparing both kinds of approaches in CI and evaluates the performance of different Machine Learning (ML) algorithms. They conclude that Ranking-to-Learn strategies are more robust regarding test case volatility, code changes, and number of failing tests. Because of this, the focus of our work is on this kind of strategy, evaluating two approaches that can be considered the state-of-the-art in TCPCI context: *RETECS*[27] (*Reinforced Test Case Selection*) and *COLEMAN*[21] (*Combinatorial Volatile Multi-Armed BAndit*).

In this sense, the work of Bertolino et al. has goals similar to ours. However, the evaluation conducted by that work uses a different approach, that is, in fact, a test case selection and prioritization approach. It includes a step to first select a test case subset before the prioritization through the learning strategies. In this way, the Ranking-to-Learn approaches were not evaluated as they were proposed in the literature. The work does not evaluate a MAB strategy, as used by *COLEMAN*. The learning is based on different features related to the test activity, which can be costly. We also used another set of evaluation measures, adopted in the test case prioritization context, leading to new findings and insights about the learning approaches. The next section describes both approaches *RETECS* and *COLEMAN*, in details.

3 LEARNING-BASED APPROACHES

This work aims to evaluate Ranking-to-Learn approaches for TCPCI problem by comparing their solutions with optimal solutions produced by a deterministic approach. Figure 2 illustrates how such approaches work in the CI environment. After a successful build, the approaches are applied before the test execution and perform the prioritization of a test case set (T) available for the current commit (c). Thus, the test cases of the prioritized test case set (T') are executed until the available test budget is reached. Feedback (reward) about T' execution is obtained and used the approaches to adapt its experience for future actions (*online learning*). In the end, a report is generated, and the developers are informed.

Next, we present the two learning-based approaches used in this work: *RETECS* and *COLEMAN*, as well as the reward functions used by both approaches in our evaluation, *RNFail* and *TimeRank*.

3.1 RETECS

RETECS (Reinforced Test Case Selection), introduced by Spieker et al. [27], is a Reinforcement Learning (RL) based approach. It uses an agent, for instance, an *Artificial Neural Network* (ANN) or a Tableau Representation, to interact with the CI environment. Based on the environment *state*, the agent defines an *action* (prioritization) to be applied in such an environment. The *state* is given by the information about a test case, such as the test case duration, historical failure data, and previous last execution. After, according to its previous action's performance, the agent receives a reward (feedback).

Based on rewards provided by a reward function, the agent adapts its experience for future actions (*online learning*). To avoid learning with irrelevant information due to long history information (low reliability with the actual behavior of the system under test), a memory representation (*sliding window*) is used to delimit how much past information is used to learn.

Spieker et al. compared different variants of RL agents and evaluated the best variation against three basic TCP methods, a random prioritization and two deterministic methods: *Sorting* and *Weighting*. The *Sorting* method prioritizes giving higher priority for the test cases that failed recently, while *Weighting* is based on the weighted sum given to the input information used as *state* in the RL agent. According to the authors, *RETECS* using the ANN variant presented the best results. For this, we evaluated the performance of *RETECS* using an ANN as an agent in our study.

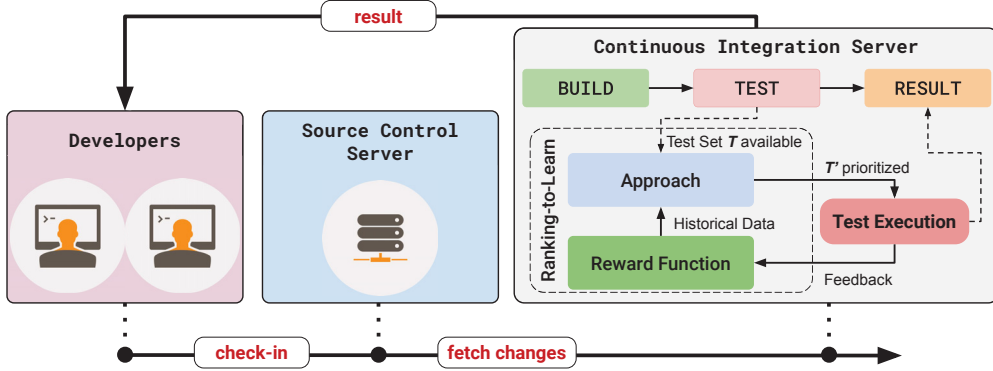


Figure 2: Integration of the evaluated learning approaches in the CI environment.

3.2 COLEMAN

COLEMAN is a *Multi-Armed Bandit* (MAB) [13] based approach design to solve the TCPPI problem dealing with the Exploration versus Exploitation (EvE) dilemma [25]. In such a dilemma, there is a balance in the search between solutions with the best performance and dissimilar solutions.

In the MAB scenario, a player plays on a set of slot machines (or arms/actions) that even identical produce different gains. After a player pulls one of the arms in a turn (c), a reward is received drawn from some unknown distribution, thus aiming to maximize the sum of the rewards. Different strategies, called MAB policies, can be used to choose the next arm by observing previous rewards and decisions.

Similarly, COLEMAN considers that a test case is an arm, but it encompasses the dynamic nature of the TCPPI problem. For this, COLEMAN incorporates two variants of MAB: volatile and combinatorial. In the first variation, the approach selects multiple arms in each turn (commit), rather than one, to produce an ordered set. In the second one, only the test cases available in each commit are considered for prioritization. The second variation aims to deal with the test case volatility. In the end, a reward function is used to obtain feedback (reward) from the prioritization proposed by the approach. Based on such feedback, the approach aims to incorporate the learning from the application of the prioritized test case set.

COLEMAN is generic and lightweight. That is, it does not require any further detail about the system under tests such as code coverage or code instrumentation, as well as, it allows the use of any MAB policy and requires only historical failure data. According to the authors of COLEMAN, it is possible to use different MAB policies. Among the policies evaluated in the experiments performed, the *Fitness-Rate-Rank based on Multi-Armed Bandit* (FRRMAB) policy [14] presented the best performance.

FRRMAB is a state policy that works with a sliding window SW as a smoother way to consider dynamic environments, allowing the observation of the changes in the quality of the arms (test cases) along the search process. The use of a SW allows evaluating a test case without it being hampered by its performance at a very early stage, which may be irrelevant to its current performance.

The FRRMAB policy was used in a further study that analyses the trade-offs of the COLEMAN solutions in comparison with the near-optimal solutions generated by a Genetic Algorithm (GA) [22]. Such an study shows that, except for one system, COLEMAN yields near-optimal solutions with negligible time. The unique exception was under a restrictive test budget associated with a few historical data. Because of this, FRRMAB is also adopted in our study that compares COLEMAN with a deterministic approach.

3.3 Reward Functions

Reward functions are used to evaluate the performance of a prioritization. In this work, we use the same functions adopted in [21]. They are adapted from the work of Spieker et al [27]. They are: Reward Based on Failures (*RNFail*) and Reward based on Time-ranked (*TimeRank*), as follows.

Let t'_c to be a test case from a prioritized test case set T'_c at a commit (cycle) c . The first reward function *RNFail* (Equation 1) is based on the number of failures associated with a test case $t'_c \in T'_c$. This function captures the ability of a test case to produce failures.

$$RNFail(t'_c) = \begin{cases} 1 & \text{if } t'_c \text{ failed} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The second function is *TimeRank*, defined in Equation 2. Let T'^{fail} is composed by the failing test cases from T'_c ; The $prec(t'_{c_1}, t'_{c_2})$ function returns 1 if the position in T'_c of t'_{c_1} is lower than the position of t'_{c_2} .

$$TimeRank(t'_c) = \frac{|T'^{fail}| - [\neg(fails(t'_c))]}{|T'^{fail}|} \times \frac{\sum_{i=1}^{|T'^{fail}|} prec(t'_c, t'_{c_i})}{|T'^{fail}|} \quad (2)$$

The *TimeRank* function observes the rank of each t'_c in T'_c and considers the problem of early scheduling. This function privileges the failing test cases ranked in the first positions in T'_c , and it penalizes those that do not fail and precede failing ones. A non-failed test case receives a reward given by the accumulated number of test cases that failed until its position in the prioritization rank.

That is, it receives a reward decreased by the number of failing test cases ranked after it.

4 EVALUATION METHODOLOGY

This section describes how the evaluation was conducted, by presenting objectives, used measures and systems, how the optimal solutions were obtained, and how *RETECS* and *COLEMAN* were executed.

The evaluation was guided by the following research question: “How far are the solutions produced by learning approaches from optimal solutions obtained through a deterministic approach?” To answer this question, we adopted a methodology following Wohlin et al.’s principles [29]. We formulated our main objectives according to the Goal Question Metric (GQM) method [2], as described in Table 1.

Table 1: Goal Question Metric Formulation

Goal:	to evaluate learning-based approaches
Purpose:	by analyzing their solutions
With respect to:	the optimal solutions generated by a deterministic approach
From the point of view:	of the tester
In the context of:	CI environments
Evaluation Measures:	fault detection effectiveness (NAPFD) fault detection effectiveness with cost consideration (APFDc) early fault detection (RFTC) test time reduction (NTR) prioritization time (PR) accuracy (RMSE)

4.1 Evaluation Measures

In our study, we used six measures. These metrics were chosen because they are largely used in the TCP literature [31]. The first one, *NAPFD* (*Normalized APFD*) metric [24] (Equation 3), evaluates fault detection effectiveness and is an extension of the *APFD Average Percentage of Faults Detected* (*APFD*) [26] metric. *APFD* indicates how quickly a set of prioritized test cases (T') can detect faults present in the application being tested. On the other hand, the *NAPFD* metric considers the ratio between detected and detectable faults within T . *NAPFD* fits the CI time constraints adequately, when not all the test cases are executed due to a time budget, and faults can be undetected. Higher *NAPFD* values indicate that the faults are detected faster using fewer test cases.

$$NAPFD(T') = p - \frac{\sum_1^n rank(T'_i)}{m \times n} \frac{p}{2n} \quad (3)$$

where m denotes the number of faults detected by all test cases; $rank(T'_i)$ is the position of T'_i in T' , if T'_i did not reveal a fault we set $T'_i = 0$; n denotes the number of test cases in T' ; and p denotes

the number of faults detected by T' divided by m . *NAPFD* is equal to *APFD* metric if all faults are detected.

The second measure, named *APFDc* (*APFD with cost consideration*) [6] (Equation 4), is also an extension from *APFD*. This metric assumes that the test cases do not have the same cost. Thus, we can consider that a test case can be more costly to execute than others, concerning, for instance, to execution time. The cost can be used as a limit, in which the test cases are usually prioritized until a maximum cost is reached. Besides that, *APFDc* can compute the *APFD* value, whether both fault severity and test case costs are identical. In this work, we consider that all faults have the same severity.

$$APFDc(T'_i) = \frac{\sum_{i=1}^m (\sum_{j=TF_i}^n c_j - 0.5c_{TF_i})}{\sum_{j=1}^n c_j \times m} \quad (4)$$

where c_i is the cost of a test case T_i , and TF_i is the first test case from T' that reveals fault i .

The third measure *Rank of the Failing Test Cases* (*RFTC*) evaluates the test suite efficiency concerning early fault detection. In such a rank, the value represents the first failing test cases’ execution order in the prioritized test suite.

We defined a fourth measure, *Normalized Time Reduction* (*NTR*) (Equation 5), to capture the difference between the time spent until the first test case fails r_t and the total time spent to execute all tests \hat{r}_t . Only the commits which failed CI^{fail} are considered in the calculation. In this way, we can evaluate the capability of an algorithm to reduce the time spent in a CI cycle.

$$NTR(\mathcal{A}) = \frac{\sum_{t=1}^{CI^{fail}} (\hat{r}_t - r_t)}{\sum_{t=1}^{CI^{fail}} (\hat{r}_t)} \quad (5)$$

The fifth measure, *Prioritization Time* (*PR*), measures the time spent by an approach to perform the prioritization. *PR* evaluates the applicability of an approach. Based on this value, we can observe whether an approach spends much time, making it impracticable for real scenarios.

The last measure, *Root-Mean-Square-Error* (*RMSE*), measures the difference between the predicted and the observed values of *NAPFD* (or *APFDc*). In our case, *RMSE* (Equation 6) is the difference between the value calculated for T' in a CI Cycle (commit) t , suggested by the learning approaches (\hat{s}_t) and the optimal value T' (s_t) found by the deterministic approach. For an algorithm \mathcal{A} , the *RMSE* is computed as follows:

$$RMSE(\mathcal{A}) = \sqrt{\frac{\sum_{t=1}^{CI} (\hat{s}_t - s_t)^2}{CI}} \quad (6)$$

where CI is the amount of CI Cycles in a system. The most accurate approach is the one with smallest *RMSE*.

We compared the results using Kruskal-Wallis [12] and Mann-Whitney [16] statistical tests to determine the significance level, and Vargha and Delaney’s \hat{A}_{12} [28] metric as effect test. The statistical tests were applied with 95% of confidence. The \hat{A}_{12} metric calculates the effect size magnitude of the difference between two groups, which defines the probability of a value taken randomly from the first sample is higher than a value taken randomly from the second sample.

The magnitude can be: *Negligible* ($\hat{A}_{12} < 0.56$); *Small* ($0.56 \leq \hat{A}_{12} < 0.64$); *Medium* ($0.64 \leq \hat{A}_{12} < 0.71$); and *Large* ($0.71 \leq \hat{A}_{12}$). A *Negligible* magnitude represents a very small difference among the values and usually does not yield statistical difference. The *Small* and *Medium* magnitudes may yield statistical differences (or not). Finally, a *Large* magnitude represents a significantly large difference that usually can be seen in the numbers without much effort.

A *Negligible* magnitude represents a very small difference among the values and usually does not yield statistical difference. The *Small* and *Medium* magnitudes represent small and medium differences among the values, and may or not yield statistical differences. Finally, a *Large* magnitude represents a significantly large difference that usually can be seen in the numbers without much effort.

4.2 Target Systems

The study was performed with twelve systems already used in the literature [21, 27, 32]. The target systems are detailed in Table 2 that contains: the system name, the period of build logs analyzed, the total of builds identified, and in parentheses, the number of builds included in the analysis. Build logs with some problems were discarded, e.g., extracting information (non-valid build log), and those the test cases did not execute. The fourth column shows the total of failures found; in parentheses, the number of builds in which at least one test failed. The fifth column shows the number of unique test cases identified from build logs; in parenthesis, the range of test cases executed in the builds. The sixth and seventh columns present the average duration and standard deviation in minutes of the CI Cycles (commits), and the interval between them.

Druid, developed by Alibaba, is a database connection pool written in Java. Fastjson, created by Alibaba, is a Java library that can be used to a fast JSON parser/generator for Java. Deeplearning4j is a deep learning library for Java Virtual Machine. DSpace is an open source software that provides facilities for the management of digital collections, used for the implementation of institutional repositories. GSDTSR is *The Google Dataset of Testing Results* [7] with a sample of 3.5 million test suite execution results from Google products. Guava, developed by Google, is a set of core libraries for Java which includes new collection types, APIs/utilities for concurrency, I/O, and others. OkHttp, developed by Square, is an HTTP and HTTP/2 client for Android and Java applications. Retrofit, also developed by Square, is a type-safe HTTP client for Android and Java. ZXing (*Zebra Crossing*) is a barcode scanning library for Java and Android. The systems IOF/ROL and Paint Control are industrial datasets for testing complex industrial robots from ABB Robotic [27]. LexisNexis is an industrial dataset for testing complex web-system at LexisNexis company [32].

More details about these systems are available in our replication package². This package contains some figures illustrating number of failures per cycle for each SUT, which allows observing test case volatility.

²Our supplementary material is available at https://osf.io/x96fk/?view_only=020b612cbdd84fa38d6a974743f9d823. After publication, we will use a DOI as a reference for the material.

4.3 Generating Optimal Solutions

The deterministic approach finds the optimal solution for each commit associated with a SUT. The failure results from the test case execution are known a priori and used in the prioritization, according to Algorithm 1.

We identify the test set available T_c and the original time A_c spent to run such set in each commit. After, we define a time budget to run the tests. In this work, we evaluated three time budgets (TB_c) concerning, respectively, 10%, 50%, and 80% of the execution time of the overall test set T_c available. They were chosen to observe the influence of the test budget in the results and how it affects the learning process, as well as they already were used in previous work [21]. In the end, we sort the test case by the number of failures in descending order, and test case duration in ascending order. This sorter allows evaluating the prioritized test set through different measures, such as failure detection (NAPFD) and cost (APFDc).

Algorithm 1: Deterministic Algorithm to Find Optimal Solutions for Test Case Prioritization in Continuous Integration Environments Problem.

```

forall commit  $c$  in Target System do
   $T_c \leftarrow$  Test Case set available from system in the current
  commit;
   $A_c \leftarrow$  Total time spent to run  $T_c$ ;
   $TB_c \leftarrow$  Time Budget (10%, 50%, or 80% from  $A_c$ );
   $T'_c \leftarrow T_c$  ordered by number of failures (descending) and
  duration (ascending);
  Evaluate  $T'_c$  considering  $TB_c$  (e.g. NAPFD and APFDc);
end

```

4.4 Executing Learning-based approaches

We use the results from the execution of *COLEMAN* and *RETECS* available in [21]³. They were obtained with 30 independent executions for each algorithm/configuration, using both reward functions, *RNFail* and *TimeRank*. *COLEMAN* was configured with FRRMAB policy, sliding window size SW equals to 100, coefficient C to balance exploration and exploitation equals to 0.3, and decayed factor equals to 1. *RETECS* was executed with an Artificial Neural Network (ANN), and the values used for Hidden Nodes, Replay Memory, and Replay Batch Size, are, respectively, 12, 10000, and 1000. All the experiments were performed on an Intel Xeon E5-2640 v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS. The system LexisNexis was not used in previous work [21]. For this system, we executed the experiments following the same settings abovementioned. The deterministic algorithm was executed in the same computational environment.

5 RESULTS AND ANALYSIS

In this section, we analyze the results of the learning approaches, *RETECS* and *COLEMAN*, having as a baseline the deterministic approach and our six measures.

³Supplementary material available at <https://doi.org/10.17605/OSF.IO/WMCBT>

Table 2: Description of the Target Systems

Name	Period	Buils	Failures	Test Cases	Duration (min)	Interval (min)
Druid	2016/04/24-2016/11/08	286 (168)	270 (71)	2391 (1778-1910)	4.97 ± 10.66	384.76 ± 468.86
Fastjson	2016/04/15-2018/12/04	2710 (2371)	940 (323)	2416 (900-2102)	1.97 ± 0.89	233.22 ± 401.26
Deeplearning4j	2014/02/22-2016/01/01	3410 (483)	777 (323)	117 (1-52)	12.33 ± 14.91	306.05 ± 442.55
DSpace	2013/10/16-2019/01/08	6309 (5673)	13413 (387)	211 (16-136)	11.78 ± 7.03	291.29 ± 411.19
GSDTSR	2016/01/02-2016/02/01	259388 (259388)	3208 (2924)	5555 (1-390)	974.25 ± 4850.66	1439.91 ± 2.58
Guava	2014/11/06-2018/12/02	2011 (1689)	7659 (112)	568 (308-512)	62.53 ± 80.31	435.55 ± 464.52
IOF/ROL	2015/02/13-2016/10/25	2392 (2392)	9289 (1627)	1941 (1-707)	1537.27 ± 2018.73	1324.36 ± 291.78
LexisNexis	2018/09/27-2018/11/15	54 (54)	21189 (54)	2662 (2007-2377)	0.8668 ± 0.808	900.367 ± 305.125
OkHttp	2013/03/26-2018/05/30	9919 (6215)	9586 (1408)	289 (2-75)	7.64 ± 5.64	220.17 ± 405.93
Paint Control	2016/01/12-2016/12/20	20711 (20711)	4956 (1980)	1980 (1-74)	424.46 ± 275.90	1417.86 ± 144.97
Retrofit	2013/02/17-2018/11/26	3719 (2711)	611 (125)	206 (5-75)	2.40 ± 1.60	270.86 ± 449.41
ZXing	2014/01/17-2017/04/16	961 (605)	68 (11)	124 (81-123)	13.14 ± 12.37	411.10 ± 465.53

5.1 Fault Detection Effectiveness

To evaluate the prioritization quality regarding fault detection capacity, we compare NAPFD average values presented in Table 3. This table presents average values \pm standard deviation. The best values are highlighted in bold. We applied the Kruskal-Wallis test to compare the algorithms regarding each measure. Results that are statistically equivalent to the best one are highlight in gray. We also use different symbols to indicate the effect size magnitude concerning the best values: “★” denotes the best algorithm for a time budget in a SUT. “▼” indicates a negligible effect size; “▽” denotes a small magnitude, “△” a medium magnitude, and “▲” large.

As expected, the deterministic approach presents the best values for all systems and budgets with statistical difference, that has, in most cases, a *large* magnitude.

Although there are statistical differences, in some cases, we observe statistical equivalence, mainly for COLEMAN using *TimeRank* function, in the less restrictive scenarios (budgets of 50% and 80%). This means the results are very close to optimal. Using *TimeRank*, for all systems and budgets, COLEMAN obtained equivalence to the optimal in 15 cases out of 36 ($\approx 42\%$). Considering each budget 10%, 50%, and 80%, COLEMAN reaches equivalence in, respectively, 17%, 50%, and 58% of the cases. COLEMAN does not reach such a good performance using *RNFail* function. In contrast, RETECS reaches results equivalent to the optimal using *RNFail* function, but only in 3 cases, out of 36 ($\approx 8\%$). Its performance seems not be impacted by the test budget. In conclusion, COLEMAN outperforms RETECS regarding early fault detection.

Finding 1. COLEMAN presents better performance than RETECS. Using *TimeRank*, COLEMAN obtained results that are equivalent to optimal in 42% of the cases. This percentage increases in the presence of less restrictive budgets, reaching 58% of the cases in the time budget of 80%.

To a better visualization, Figure 3 illustrates radar charts (or spider graphs) for each time budget. Each angle represents the NAPFD value for a system. The purple line represents the values found by the *Deterministic* approach; blue and orange lines obtained

by RETECS using, respectively, *RNFail* and *TimeRank* functions; and green and red lines obtained by COLEMAN using *RNFail* and *TimeRank* functions.

As we can observe, increasing the time budget, the learning-based approaches produce solutions closer to optimal, mainly COLEMAN. In some systems, we observe that, even increasing the time budget, the difference keeps the same; for instance, Deeplearning4j and OkHttp. To a deeper analysis, we refer to Figure 4. This figure presents, in overall (in the same scale), different information about each system: number of valid builds, number of failures, number of failed builds, number of test cases, mean number of failures by cycles, and mean number of failing cycles. More information is found in Table 2.

Regarding the Deeplearning4j system, it has a high average of failing builds. However, as we observed, only this does not help to provide good prioritization. On the other hand, OkHttp has a small average of failing builds but more failures and failed builds. This helped to provide better NAPFD values than in the Deeplearning4j system. In both systems, COLEMAN obtained equivalence to optimal in all time budgets evaluated.

We analyzed other characteristics of the systems that may impact the prioritization, e.g., the test case volatility. We observe in the Deeplearning4j and OkHttp systems that the failures are frequent and well distributed in some tests, even having peaks of failures and test case volatility. This scenario endorses an approach based on historical test data.

Among the systems, ZXing is the simplest one. In this system, the values found by learning-based approaches are the closest to the optimal, that is, close to 1 which is the maximum value for NAPFD. There is low test case volatility in this system, and there are peaks in the failure detection in a few commits, with long periods without failures. This situation can also be supported by an approach based on historical test data.

Regarding the NAPFD results that are equivalent to the optimal, we observe that *RNFail* fits better with RETECS and *TimeRank* with COLEMAN. The worst results were obtained by RETECS and COLEMAN in the Druid and LexisNexis systems. The Druid system presents the greatest difference between learning-based approaches and *Deterministic*, mainly in the presence of the most restrictive

Table 3: NAPFD comparison.

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
TIME BUDGET: 10%					
Druid	0.6768 ± 0.129 ▲	0.6488 ± 0.074 ▲	0.9996 ± 0.000 ★	0.6801 ± 0.052 ▲	0.7137 ± 0.074 ▲
Fastjson	0.8713 ± 0.015 ▲	0.8719 ± 0.005 ▲	0.9988 ± 0.000 ★	0.9030 ± 0.014 ▲	0.8980 ± 0.018 ▲
Deeplearning4j	0.6615 ± 0.072 ▲	0.6739 ± 0.016 ▲	0.8137 ± 0.000 ★	0.7533 ± 0.002 ▲	0.7716 ± 0.000 ▲
DSpace	0.9437 ± 0.001 ▲	0.9410 ± 0.001 ▲	0.9739 ± 0.000 ★	0.9489 ± 0.003 ▲	0.9496 ± 0.004 ▲
GSDTSR	0.9893 ± 0.000 ▲	0.9893 ± 0.000 ▲	0.9898 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9676 ± 0.015 ▲	0.9563 ± 0.004 ▲	0.9978 ± 0.000 ★	0.9554 ± 0.002 ▲	0.9586 ± 0.001 ▲
IOF/ROL	0.3704 ± 0.005 ▲	0.3779 ± 0.003 ▲	0.4098 ± 0.000 ★	0.3632 ± 0.001 ▲	0.3670 ± 0.001 ▲
LexisNexis	0.0508 ± 0.018 ▲	0.1004 ± 0.068 ▲	0.7011 ± 0.000 ★	0.1400 ± 0.001 ▲	0.1440 ± 0.001 ▲
Paint Control	0.9078 ± 0.000 ▼	0.9077 ± 0.000 ▲	0.9078 ± 0.000 ★	0.9076 ± 0.000 ▲	0.9076 ± 0.000 ▲
OkHttp	0.8357 ± 0.002 ▲	0.8095 ± 0.006 ▲	0.8886 ± 0.000 ★	0.8323 ± 0.000 ▲	0.8407 ± 0.000 ▲
Retrofit	0.9641 ± 0.001 ▲	0.9621 ± 0.001 ▲	0.9712 ± 0.000 ★	0.9639 ± 0.000 ▲	0.9642 ± 0.000 ▲
Zxing	0.9854 ± 0.000 ▲	0.9855 ± 0.000 ▲	0.9998 ± 0.000 ★	0.9826 ± 0.000 ▲	0.9828 ± 0.000 ▲
TIME BUDGET: 50%					
Druid	0.6851 ± 0.134 ▲	0.6323 ± 0.074 ▲	0.9996 ± 0.000 ★	0.9333 ± 0.013 ▲	0.9710 ± 0.008 ▲
Fastjson	0.8714 ± 0.007 ▲	0.8902 ± 0.013 ▲	0.9993 ± 0.000 ★	0.9174 ± 0.021 ▲	0.9118 ± 0.028 ▲
Deeplearning4j	0.7049 ± 0.070 ▲	0.6562 ± 0.018 ▲	0.9025 ± 0.000 ★	0.7890 ± 0.001 ▲	0.8200 ± 0.000 ▲
DSpace	0.9568 ± 0.001 ▲	0.9485 ± 0.001 ▲	0.9921 ± 0.000 ★	0.9724 ± 0.009 ▲	0.9766 ± 0.008 ▲
GSDTSR	0.9911 ± 0.000 ▲	0.9906 ± 0.000 ▲	0.9921 ± 0.000 ★	0.9893 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9502 ± 0.015 ▲	0.9578 ± 0.004 ▲	0.9997 ± 0.000 ★	0.9653 ± 0.004 ▲	0.9675 ± 0.007 ▲
IOF/ROL	0.5101 ± 0.007 ▲	0.5025 ± 0.006 ▲	0.5812 ± 0.000 ★	0.5046 ± 0.002 ▲	0.5189 ± 0.002 ▲
LexisNexis	0.1629 ± 0.026 ▲	0.2335 ± 0.099 ▲	0.9065 ± 0.000 ★	0.5332 ± 0.007 ▲	0.5625 ± 0.008 ▲
Paint Control	0.9150 ± 0.000 ▲	0.9138 ± 0.000 ▲	0.9153 ± 0.000 ★	0.9150 ± 0.000 ▲	0.9150 ± 0.000 ▲
OkHttp	0.8812 ± 0.010 ▲	0.8446 ± 0.003 ▲	0.9544 ± 0.000 ★	0.9192 ± 0.000 ▲	0.9317 ± 0.000 ▲
Retrofit	0.9706 ± 0.002 ▲	0.9718 ± 0.002 ▲	0.9946 ± 0.000 ★	0.9853 ± 0.000 ▲	0.9893 ± 0.000 ▲
Zxing	0.9878 ± 0.000 ▲	0.9881 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9846 ± 0.000 ▲	0.9857 ± 0.000 ▲
TIME BUDGET: 80%					
Druid	0.6490 ± 0.113 ▲	0.6551 ± 0.099 ▲	0.9996 ± 0.000 ★	0.938 ± 0.012 ▲	0.9830 ± 0.003 ▲
Fastjson	0.8708 ± 0.007 ▲	0.8925 ± 0.010 ▲	0.9999 ± 0.000 ★	0.9536 ± 0.010 ▲	0.9242 ± 0.028 ▲
Deeplearning4j	0.7058 ± 0.091 ▲	0.6640 ± 0.016 ▲	0.9520 ± 0.000 ★	0.8424 ± 0.001 ▲	0.8641 ± 0.001 ▲
DSpace	0.9601 ± 0.001 ▲	0.9508 ± 0.001 ▲	0.9932 ± 0.000 ★	0.9792 ± 0.006 ▲	0.9825 ± 0.007 ▲
GSDTSR	0.9921 ± 0.000 ▲	0.9914 ± 0.000 ▲	0.9934 ± 0.000 ★	0.9893 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9441 ± 0.012 ▲	0.9581 ± 0.007 ▲	0.9999 ± 0.000 ★	0.9784 ± 0.012 ▲	0.9841 ± 0.014 ▲
IOF/ROL	0.5495 ± 0.006 ▲	0.5287 ± 0.007 ▲	0.6115 ± 0.000 ★	0.5569 ± 0.002 ▲	0.5678 ± 0.001 ▲
LexisNexis	0.2496 ± 0.048 ▲	0.3545 ± 0.131 ▲	0.9152 ± 0.000 ★	0.6442 ± 0.005 ▲	0.7033 ± 0.004 ▲
Paint Control	0.9162 ± 0.000 ▲	0.9160 ± 0.000 ▲	0.9180 ± 0.000 ★	0.9171 ± 0.000 ▲	0.9171 ± 0.000 ▲
OkHttp	0.9027 ± 0.013 ▲	0.8558 ± 0.004 ▲	0.9607 ± 0.000 ★	0.935 ± 0.000 ▲	0.9478 ± 0.000 ▲
Retrofit	0.9724 ± 0.005 ▲	0.9745 ± 0.003 ▲	0.9972 ± 0.000 ★	0.9881 ± 0.000 ▲	0.9916 ± 0.000 ▲
ZXing	0.9878 ± 0.000 ▲	0.9883 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9972 ± 0.000 ▲	0.9996 ± 0.000 ▲

time budget (10%). These systems share some particularities: (i) a few number of CI Cycles; and (ii) a large test case set, in which many failures are distributed in many test cases. Apparently, such characteristics are drawbacks for approaches based on historical test data.

Finding 2. The learning-based approaches have the worst performance in systems with a high test case volatility and a few number of CI Cycles.

Concerning the optimal results, some of them are far from the maximum value for the metric, specifically for the systems IOF/ROL and LexisNexis. About the IOF/ROL system, we observe that the difficulty in obtaining better NAPFD values is related to the extremely high test case volatility coupled with the high number of failures, as well as the failure distribution over many test cases. This hampers to find a reasonable prioritization. On the other hand, in the LexisNexis system, we do not observe high test case volatility but, similarly to IOF/ROL, a high number of failures distributed in many test cases. In this way, both systems are examples of why it is hard to find reasonable solutions for TCP in the CI environments.

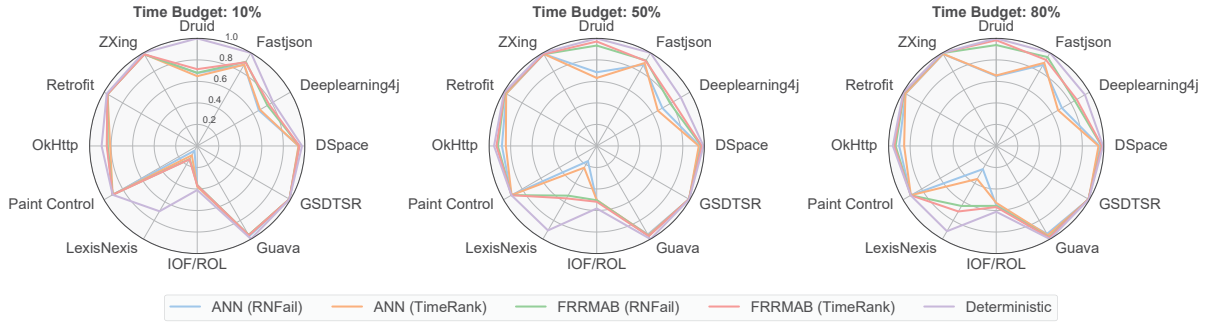


Figure 3: Radar charts - NAPFD values.

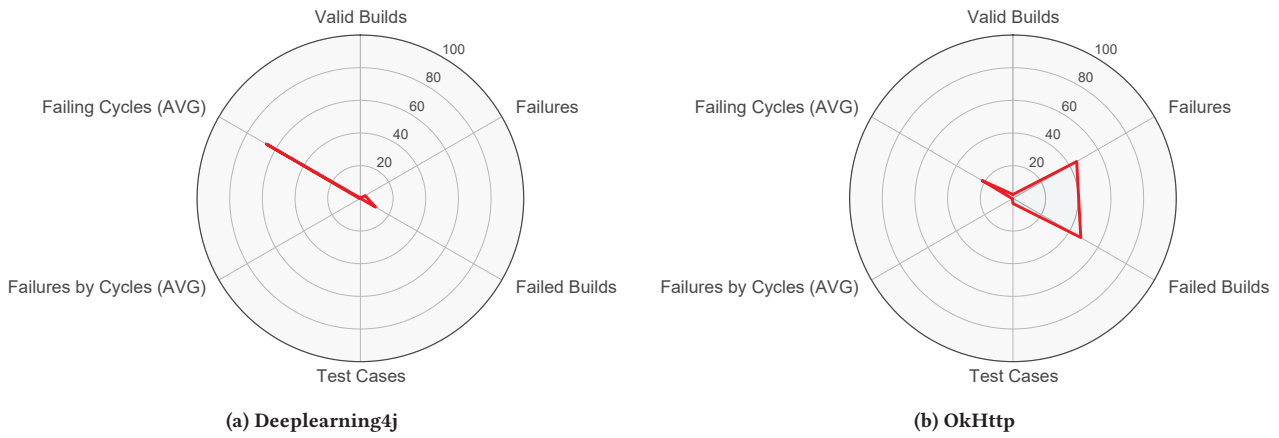


Figure 4: Radar charts - systems characteristics

Finding 3. A high number of failures distributed over many test cases makes the TCP task harder.

Finding 4. Regarding APFDc values *COLEMAN* outperforms *RETECS* in most cases. However, *RETECS* has a better performance in the most restrictive budget of 10%.

5.2 Fault Detection Effectiveness with Cost Consideration

To evaluate how good is the prioritization considering the cost associated for each test case, we calculate APFDc values, presented in Table 4. We use the test case duration as cost. If two test cases reveal the same number of failures and have different execution times, that one that takes less time needs to appear first in the prioritization rank.

As it happens for NAPFD, *COLEMAN* performs better in the less restrictive budget using *TimeRank*, reaching results equivalent to the optimal in 50% of the cases for the time budget of 80%. *RETECS* also performs better using *RNFail*, but we observe a better performance of *RETECS*, overcoming *COLEMAN*, in the more restrictive budget of 10%. This is maybe due to the *RETECS* formulation that considers test case duration during its prioritizations. Nevertheless, considering a general case, *COLEMAN* outperforms *RETECS*, even only focusing on historical failure data.

Again, radar charts regarding APFDc values can provide a better analysis (Figure 5). We can see that it is harder to obtain good prioritizations with less cost for LexisNexis for both approaches. We observe a great difference between *COLEMAN* and *RETECS* in the Druid system and the time budgets of 50% and 80%. As we observed in the NAPFD values, *COLEMAN* also has better performance with *TimeRank* function, whilst *RETECS* with *RNFail*. Besides that, NAPFD and APFDc values are not so far from optimal solutions, in which we can observe close values but with a statistical difference.

Finding 5. The analysis of APFDc using test case duration as cost leads to results similar to those obtained in the NAPFD analysis. In general, the APFDc values are close to the optimal, and the more significant differences are obtained to the same systems.

Table 4: APFDc comparison.

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
TIME BUDGET: 10%					
Druid	0.8067 ± 0.088 ▲	0.7815 ± 0.051 ▲	0.9998 ± 0.000 ★	0.6964 ± 0.063 ▲	0.7181 ± 0.076 ▲
Fastjson	0.8805 ± 0.014 ▲	0.8810 ± 0.004 ▲	0.9985 ± 0.000 ★	0.9064 ± 0.013 ▲	0.8974 ± 0.018 ▲
Deeplearning4j	0.8135 ± 0.023 ▲	0.8185 ± 0.010 ▲	0.8304 ± 0.000 ★	0.7766 ± 0.001 ▲	0.7773 ± 0.000 ▲
DSpace	0.9480 ± 0.001 ▲	0.9458 ± 0.001 ▲	0.9724 ± 0.000 ★	0.9510 ± 0.002 ▲	0.9513 ± 0.004 ▲
GSDTSR	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲	0.9898 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9811 ± 0.007 ▲	0.9761 ± 0.003 ▲	0.9980 ± 0.000 ★	0.9561 ± 0.001 ▲	0.9582 ± 0.001 ▲
IOF/ROL	0.3746 ± 0.005 ▲	0.3819 ± 0.003 ▲	0.4138 ± 0.000 ★	0.3661 ± 0.001 ▲	0.3701 ± 0.001 ▲
LexisNexis	0.0541 ± 0.017 ▲	0.1025 ± 0.066 ▲	0.7076 ± 0.000 ★	0.1394 ± 0.001 ▲	0.1434 ± 0.001 ▲
Paint Control	0.9082 ± 0.000 ▲	0.9081 ± 0.000 ▲	0.9082 ± 0.000 ★	0.9080 ± 0.000 ▲	0.9080 ± 0.000 ▲
OkHttp	0.8484 ± 0.001 ▲	0.8292 ± 0.006 ▲	0.8870 ± 0.000 ★	0.8378 ± 0.000 ▲	0.8425 ± 0.000 ▲
Retrofit	0.9672 ± 0.001 ▲	0.9655 ± 0.001 ▲	0.9717 ± 0.000 ★	0.9646 ± 0.000 ▲	0.9648 ± 0.000 ▲
ZXing	0.9893 ± 0.000 ▲	0.9893 ± 0.000 ▲	0.9998 ± 0.000 ★	0.9832 ± 0.000 ▲	0.9835 ± 0.000 ▲
TIME BUDGET: 50%					
Druid	0.8147 ± 0.102 ▲	0.7597 ± 0.051 ▲	0.9998 ± 0.000 ★	0.9486 ± 0.016 ▲	0.9787 ± 0.009 ▲
Fastjson	0.9326 ± 0.005 ▲	0.9392 ± 0.017 ▲	0.9989 ± 0.000 ★	0.9186 ± 0.021 ▲	0.9140 ± 0.027 ▲
Deeplearning4j	0.8331 ± 0.041 ▲	0.8379 ± 0.012 ▲	0.9077 ± 0.000 ★	0.8106 ± 0.001 ▲	0.8134 ± 0.001 ▲
DSpace	0.9683 ± 0.001 ▲	0.9615 ± 0.001 ▲	0.9918 ± 0.000 ★	0.9737 ± 0.009 ▲	0.9767 ± 0.009 ▲
GSDTSR	0.9911 ± 0.000 ▲	0.9910 ± 0.000 ▲	0.9920 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9767 ± 0.008 ▲	0.9806 ± 0.005 ▲	0.9993 ± 0.000 ★	0.9687 ± 0.003 ▲	0.9672 ± 0.007 ▲
IOF/ROL	0.5175 ± 0.008 ▲	0.5043 ± 0.006 ▲	0.5905 ± 0.000 ★	0.5081 ± 0.002 ▲	0.5223 ± 0.002 ▲
LexisNexis	0.1891 ± 0.019 ▲	0.2477 ± 0.093 ▲	0.9035 ± 0.000 ★	0.5227 ± 0.007 ▲	0.5496 ± 0.007 ▲
Paint Control	0.9171 ± 0.000 ▲	0.9140 ± 0.000 ▲	0.9174 ± 0.000 ★	0.9162 ± 0.000 ▲	0.9162 ± 0.000 ▲
OkHttp	0.8878 ± 0.015 ▲	0.8869 ± 0.002 ▲	0.9477 ± 0.000 ★	0.9177 ± 0.000 ▲	0.9246 ± 0.000 ▲
Retrofit	0.9762 ± 0.002 ▲	0.9778 ± 0.002 ▲	0.9928 ± 0.000 ★	0.9850 ± 0.000 ▲	0.9885 ± 0.000 ▲
ZXing	0.9954 ± 0.000 ▲	0.9956 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9862 ± 0.000 ▲	0.9869 ± 0.000 ▲
TIME BUDGET: 80%					
Druid	0.7142 ± 0.111 ▲	0.6881 ± 0.090 ▲	0.9998 ± 0.000 ★	0.9469 ± 0.015 ▲	0.9912 ± 0.004 ▲
Fastjson	0.9037 ± 0.008 ▲	0.9133 ± 0.015 ▲	0.9991 ± 0.000 ★	0.9488 ± 0.012 ▲	0.9270 ± 0.026 ▲
Deeplearning4j	0.8158 ± 0.064 ▲	0.8522 ± 0.012 ▲	0.9407 ± 0.000 ★	0.8068 ± 0.002 ▲	0.7989 ± 0.001 ▲
DSpace	0.9738 ± 0.001 ▲	0.9639 ± 0.001 ▲	0.9925 ± 0.000 ★	0.9796 ± 0.006 ▲	0.9810 ± 0.008 ▲
GSDTSR	0.9919 ± 0.000 ▲	0.9917 ± 0.000 ▲	0.9930 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9627 ± 0.010 ▲	0.9689 ± 0.009 ▲	0.9994 ± 0.000 ★	0.9780 ± 0.013 ▲	0.9825 ± 0.015 ▲
IOF/ROL	0.5593 ± 0.006 ▲	0.5311 ± 0.006 ▲	0.6225 ± 0.000 ★	0.5591 ± 0.002 ▲	0.5699 ± 0.001 ▲
LexisNexis	0.2886 ± 0.039 ▲	0.3569 ± 0.104 ▲	0.9111 ± 0.000 ★	0.6287 ± 0.005 ▲	0.6791 ± 0.004 ▲
Paint Control	0.9187 ± 0.000 ▲	0.9158 ± 0.000 ▲	0.9204 ± 0.000 ★	0.9176 ± 0.000 ▲	0.9177 ± 0.000 ▲
OkHttp	0.8836 ± 0.020 ▲	0.8974 ± 0.003 ▲	0.9520 ± 0.000 ★	0.9271 ± 0.000 ▲	0.9362 ± 0.000 ▲
Retrofit	0.9785 ± 0.004 ▲	0.9808 ± 0.003 ▲	0.9946 ± 0.000 ★	0.9873 ± 0.000 ▲	0.9903 ± 0.000 ▲
ZXing	0.9953 ± 0.000 ▲	0.9953 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9975 ± 0.000 ▲	0.9996 ± 0.000 ▲

5.3 Early Fault Detection and Test Time Reduction

First of all, we calculate RFTC values (Table 5) that takes into account the position of the first failing test case in the prioritized test case set. We observed that the NAPFD average values found and the early fault detection (using RFTC) are correlated, that is, good NAPFD values provide good RFTC values. However, the opposite can not be true, once that the RFTC metric does not evaluate the prioritization quality from the entire prioritized test set but only the early fault detection.

As expected, the deterministic approach presents the best results for all systems. Besides that, *COLEMAN* using *TimeRank* obtained

equivalent results in $\approx 70\%$ of the cases, whilst *RETECS* only in $\approx 3\%$ (only one case). In some cases, *RETECS* has a higher standard deviation than *COLEMAN*.

Finding 6. Regarding RFTC, *COLEMAN* using *TimeRank* obtained performance equivalent to the optimal results in $\approx 70\%$ of the cases. *COLEMAN* is better than *RETECS* in all cases.

Early fault detection contributes to reduce test execution cost because the test can be ended when a failure occurs. Given this fact,

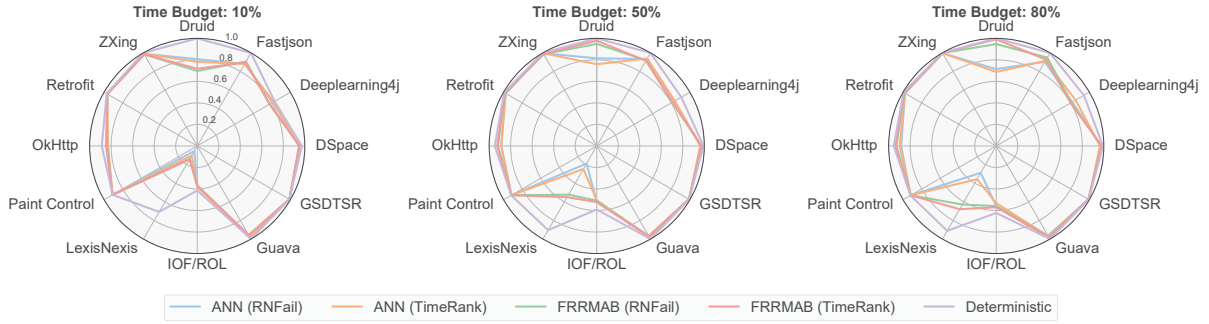


Figure 5: Radar charts - APFDc values.

Table 5: RFTC comparison.

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
TIME BUDGET: 10%					
Druid	1166.4411 ± 546.049 ▲	1240.8713 ± 356.817 ▲	1.0 ± 0.000 ★	209.3289 ± 98.987 ▲	56.1579 ± 31.056 ▲
Fastjson	1094.1147 ± 209.106 ▲	998.7949 ± 177.738 ▲	1.0 ± 0.000 ★	184.9911 ± 65.066 ▲	96.0378 ± 36.22 ▲
Deeplearning4j	5.3151 ± 2.456 ▲	5.7919 ± 0.677 ▲	1.0 ± 0.000 ★	2.3049 ± 0.048 ▲	2.0437 ± 0.007 ▲
DSpace	11.9561 ± 0.753 ▲	13.6276 ± 0.754 ▲	1.0 ± 0.000 ★	3.024 ± 1.33 ▲	2.1428 ± 0.67 ▲
GSDTSR	3.2553 ± 0.387 ▲	3.8958 ± 0.18 ▲	1.0 ± 0.000 ★	2.1316 ± 0.1 ▲	1.1482 ± 0.071 ▲
Guava	129.3477 ± 99.443 ▲	191.1219 ± 29.545 ▲	1.0 ± 0.000 ★	31.8991 ± 20.546 ▲	15.0977 ± 13.75 ▲
IOF/ROL	1.6445 ± 0.158 ▲	1.5639 ± 0.108 ▲	1.0 ± 0.000 ★	1.2626 ± 0.064 ▲	1.0992 ± 0.05 ▲
LexisNexis	73.097 ± 31.648 ▲	53.1777 ± 38.754 ▲	1.0 ± 0.000 ★	9.1333 ± 0.188 ▲	8.7611 ± 0.118 ▲
Paint Control	1.0 ± 0.000 ▼	1.001 ± 0.002 ▽	1.0 ± 0.000 ★	1.0 ± 0.000 ▼	1.0 ± 0.000 ▼
OkHttp	7.4112 ± 0.397 ▲	15.8935 ± 1.951 ▲	1.0 ± 0.000 ★	4.3424 ± 0.000 ▲	1.8039 ± 0.000 ▲
Retrofit	3.7352 ± 0.439 ▲	4.7297 ± 0.772 ▲	1.0 ± 0.000 ★	1.5152 ± 0.000 ▲	1.7941 ± 0.000 ▲
ZXing	39.7929 ± 0.643 ▲	39.1204 ± 1.701 ▲	1.0 ± 0.000 ★	1.0 ± 0.000 ▼	1.0 ± 0.000 ▼
TIME BUDGET: 50%					
Druid	1225.6197 ± 600.746 ▲	1420.3143 ± 418.926 ▲	1.0 ± 0.000 ★	121.8396 ± 43.763 ▲	51.2697 ± 11.926 ▲
Fastjson	1527.9434 ± 93.004 ▲	1173.9871 ± 321.789 ▲	1.0 ± 0.000 ★	315.8923 ± 79.836 ▲	335.9929 ± 125.629 ▲
Deeplearning4j	5.0267 ± 1.827 ▲	7.3264 ± 0.53 ▲	1.0 ± 0.000 ★	2.5496 ± 0.011 ▲	2.464 ± 0.005 ▲
DSpace	19.0354 ± 0.887 ▲	27.6301 ± 0.921 ▲	1.0 ± 0.000 ★	5.5312 ± 1.939 ▲	4.1089 ± 1.769 ▲
GSDTSR	1.9072 ± 0.06 ▲	3.5648 ± 0.141 ▲	1.0 ± 0.000 ★	2.1461 ± 0.101 ▲	1.1505 ± 0.072 ▲
Guava	289.1586 ± 99.054 ▲	235.0919 ± 27.865 ▲	1.0 ± 0.000 ★	78.6409 ± 45.928 ▲	24.0869 ± 21.479 ▲
IOF/ROL	1.8009 ± 0.292 ▲	1.9213 ± 0.218 ▲	1.0 ± 0.000 ★	1.2588 ± 0.035 ▲	1.151 ± 0.024 ▲
LexisNexis	73.6444 ± 41.192 ▲	47.3883 ± 37.993 ▲	1.0 ± 0.000 ★	9.2074 ± 0.178 ▲	8.784 ± 0.091 ▲
Paint Control	1.0234 ± 0.004 ▲	1.0257 ± 0.007 ▲	1.0 ± 0.000 ★	1.0018 ± 0.001 ▲	1.0014 ± 0.001 ▲
OkHttp	6.203 ± 2.066 ▲	19.6306 ± 0.79 ▲	1.0 ± 0.000 ★	4.188 ± 0.014 ▲	2.3643 ± 0.002 ▲
Retrofit	5.4302 ± 0.43 ▲	5.625 ± 0.415 ▲	1.0 ± 0.000 ★	2.4059 ± 0.000 ▲	1.4299 ± 0.000 ▲
ZXing	51.2576 ± 0.579 ▲	49.4232 ± 3.15 ▲	1.0 ± 0.000 ★	5.6 ± 0.000 ▲	2.0 ± 0.000 ▲
TIME BUDGET: 80%					
Druid	1427.8103 ± 493.429 ▲	1035.3369 ± 658.042 ▲	1.0 ± 0.000 ★	146.9112 ± 46.436 ▲	50.3805 ± 11.25 ▲
Fastjson	1535.2998 ± 101.625 ▲	993.382 ± 393.441 ▲	1.0 ± 0.000 ★	398.4345 ± 105.27 ▲	572.0954 ± 221.573 ▲
Deeplearning4j	5.5748 ± 2.358 ▲	7.8533 ± 0.581 ▲	1.0 ± 0.000 ★	2.8146 ± 0.014 ▲	2.5017 ± 0.011 ▲
DSpace	23.126 ± 1.021 ▲	33.4617 ± 1.119 ▲	1.0 ± 0.000 ★	6.8651 ± 2.946 ▲	6.0794 ± 3.343 ▲
GSDTSR	1.9413 ± 0.322 ▲	3.4858 ± 0.112 ▲	1.0 ± 0.000 ★	2.1461 ± 0.101 ▲	1.1505 ± 0.072 ▲
Guava	330.5342 ± 74.0 ▲	202.2301 ± 47.258 ▲	1.0 ± 0.000 ★	84.6856 ± 34.075 ▲	83.9989 ± 78.446 ▲
IOF/ROL	2.021 ± 0.447 ▲	2.5248 ± 0.362 ▲	1.0 ± 0.000 ★	1.317 ± 0.026 ▲	1.2366 ± 0.017 ▲
LexisNexis	47.6333 ± 43.432 ▲	43.5358 ± 39.71 ▲	1.0 ± 0.000 ★	9.2 ± 0.208 ▲	8.7981 ± 0.11 ▲
Paint Control	1.015 ± 0.002 ▲	1.0344 ± 0.009 ▲	1.0 ± 0.000 ★	1.0003 ± 0.001 ▽	1.0003 ± 0.001 ▽
OkHttp	4.2988 ± 2.242 ▲	21.5784 ± 1.148 ▲	1.0 ± 0.000 ★	4.0748 ± 0.021 ▲	2.282 ± 0.003 ▲
Retrofit	5.9252 ± 0.884 ▲	5.8832 ± 0.587 ▲	1.0 ± 0.000 ★	2.4636 ± 0.000 ▲	1.4386 ± 0.000 ▲
ZXing	51.0061 ± 1.233 ▲	48.6848 ± 2.926 ▲	1.0 ± 0.000 ★	4.2727 ± 0.000 ▲	1.3636 ± 0.000 ▲

we analyze NTR values (Table 6) to evaluate the impact in the time reduction inside a CI Cycle.

We can observe that in most cases, the greater the time budget the greater the NTR values. The deterministic approach gets the best reduction values. However, other approaches have close values for most systems. The best percentages of time reduction for *COLEMAN* considering *TimeRank* function are in the systems: LexisNexis with 99.61% in all time budgets; and IOF/ROL with 57.01%, 71.93%, and 73.94%, for the time budgets, respectively of, 10%, 50%, and 80%. On the other hand, *RETECS* presents the best values considering *RNFail* function in *Deeplearning4j*, with 55.46%, 54.47%, and 54.75%, respectively for three budgets.

As mentioned before, for LexisNexis and IOF/ROL the failures are distributed over many test cases. This corroborates to the early fault detection once that there is a high probability of prioritizing a failing test case in the first positions.

The percentage found by the deterministic approach is low for some systems, such as Guava, Retrofit, and ZXing. In these systems, there is a low failure distribution across the test cases along with peaks of failures in a few CI Cycles, and the failing test cases vary in each CI Cycle. This shows that sometimes we face test cases that fail but spend much time executing, and there is not a pattern that hampers a reasonable prioritization using historical failure data.

Finding 7. Even using a deterministic approach, sometimes, the test time reduction is low due to peaks of failures, failure distributed across the test cases, and variation of the failing test cases over CI cycles. Nevertheless, *COLEMAN* reached high percentages of reductions for systems, considered hard cases for prioritizing, such as LexisNexis.

5.4 Prioritization Time

We also observed the time spent to prioritize the test cases (Prioritization Time in Table 7). Although the deterministic approach is only a simple order, it can be used as a baseline.

We can observe the time spent by the approaches is negligible, even whimsy in most systems. A great time is spent in Druid, Fastjson, and LexisNexis systems that also have a significant number of test cases in a CI Cycle. *RETECS* using *RNFail* function has PR values that are statistically equivalent to the optimal in 23 ($\approx 73\%$) cases out of 36. In three cases, it presents the best values for system *Paint Control*. But *RETECS* also presents the greatest variations; see system LexisNexis. In overall, *RETECS* and *COLEMAN* spend less than one second to perform the prioritization.

To observe the applicability in real scenarios, we considered the information presented in Table 2 regarding each SUT. In such a table, we present the time spent in a CI Cycle and the interval between commits for each system. As we can observe, typically, a new commit is performed, with a considered time, after a CI Cycle is ended. Such systems do not present a situation with multiple test requests, except in IOF/ROL system. As mentioned before, the approaches can reduce $\approx 99\%$ of the CI Cycle time in such a system.

Moreover, the time presented in Table 2 is in minutes while the prioritization time of the approaches is presented in Table 7 in seconds. In this way, considering the interval between CI cycles, there is no negative impact in the use of the approaches. Furthermore, they can help developers concerning the time they spend waiting for test feedback.

Finding 8. The learning-based approaches are applicable in our real scenarios. Overall, they spend less than one second to execute.

5.5 Accuracy

The accuracy (RMSE) is given by the difference between the predicted and the observed values of NAPFD and APFDc; these last ones are obtained by the deterministic approach. The results are presented in Tables 8 and 9. By analyzing such tables we can corroborate our previous findings.

Regarding the RMSE values for NAPFD metric (Table 8), we observe the predominance of *COLEMAN* (using *TimeRank* function) against *RETECS*. In the presence of a restrictive time budget (10%), *RETECS* performs better than in the other ones. However, *COLEMAN* is better in all time budgets.

The learning-based approaches obtain small RMSE values in almost all systems, except in Druid and LexisNexis. The smallest RMSE values are obtained under time budget of 80% and by *COLEMAN* for the systems Druid, DSpace, Guava, Paint Control, OkHttp, Retrofit, and ZXing. In these systems, the NAPFD values obtained are the closest to the optimal values. To a better visualization, we generated Figure 6.

One interesting point is that RMSE values lower than 0.2 represent NAPFD values closer to optimal values. For instance, in IOF/ROL system, the NAPFD values are low, but this is because it is challenging to find reasonable solutions. On the other hand, in the LexisNexis system, the deterministic approach also obtained low NAPFD values, but the learning-based approaches obtained the worst RMSE values. Such values are between 0.22 and 0.76.

RETECS gets the worst RMSE values for the systems Druid and LexisNexis in the time budget of 50%. They are, respectively, $RMSE \geq 0.58$ and $RMSE \geq \approx 0.75$. For these systems, this phenomenon occurs, as mentioned before, due to the lack of historical data. Besides them, we can observe in Figure 6 that the learning-based approaches also have a poor performance for *Deeplearning4j*. In this system, we do not find a correlation between test case volatility and the number of failures. For this, we investigated the accumulative NAPFD across the CI Cycles (Figure 7).

As we can observe, the NAPFD values change a bit before the 100th CI Cycle and normalize after the 300th. Near to the 100th cycle, the system *Deeplearning4j* starts presenting more failures, and the duration of some test cases starts increasing. Probably, such behavior influences more decisions taken by *RETECS* than by *COLEMAN*, once the first considers besides the failures, the test case duration. From then on, the number of failures increases with the test case volatility. This may have favored a catastrophic forgetting in the ANN.

Table 6: NTR comparison

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
TIME BUDGET: 10%					
Druid	0.1863 ± 0.124	0.1556 ± 0.077	0.4331 ± 0.000	0.2027 ± 0.115	0.2355 ± 0.135
Fastjson	0.0194 ± 0.016	0.0219 ± 0.007	0.1442 ± 0.000	0.0681 ± 0.014	0.0566 ± 0.020
Deeplearning4j	0.5488 ± 0.029	0.5546 ± 0.015	0.5642 ± 0.000	0.4626 ± 0.001	0.4663 ± 0.000
DSpace	0.0188 ± 0.001	0.0105 ± 0.001	0.0476 ± 0.000	0.0239 ± 0.002	0.0251 ± 0.003
GSDTSR	0.0087 ± 0.000	0.0090 ± 0.000	0.0136 ± 0.000	0.0093 ± 0.000	0.0096 ± 0.000
Guava	0.0449 ± 0.015	0.0367 ± 0.005	0.0660 ± 0.000	0.0313 ± 0.001	0.0333 ± 0.002
IOF/ROL	0.5133 ± 0.030	0.5646 ± 0.015	0.6222 ± 0.000	0.5585 ± 0.007	0.5701 ± 0.004
LexisNexis	0.9784 ± 0.012	0.9863 ± 0.010	0.9999 ± 0.000	0.9960 ± 0.000	0.9961 ± 0.000
Paint Control	0.1151 ± 0.000	0.1139 ± 0.000	0.1152 ± 0.000	0.1133 ± 0.000	0.1131 ± 0.000
OkHttp	0.0830 ± 0.001	0.0579 ± 0.008	0.1160 ± 0.000	0.0658 ± 0.000	0.0702 ± 0.000
Retrofit	0.0088 ± 0.000	0.0076 ± 0.001	0.0126 ± 0.000	0.0070 ± 0.000	0.0073 ± 0.000
ZXing	0.0122 ± 0.000	0.0126 ± 0.001	0.0227 ± 0.000	0.0037 ± 0.000	0.0037 ± 0.000
TIME BUDGET: 50%					
Druid	0.1840 ± 0.137	0.1213 ± 0.071	0.4331 ± 0.000	0.4057 ± 0.013	0.4225 ± 0.008
Fastjson	0.0399 ± 0.010	0.0602 ± 0.020	0.1445 ± 0.000	0.0768 ± 0.018	0.0724 ± 0.023
Deeplearning4j	0.5276 ± 0.039	0.5447 ± 0.010	0.5788 ± 0.000	0.4695 ± 0.000	0.4625 ± 0.000
DSpace	0.0334 ± 0.001	0.0218 ± 0.002	0.0604 ± 0.000	0.0486 ± 0.004	0.0499 ± 0.006
GSDTSR	0.0199 ± 0.000	0.0179 ± 0.000	0.0259 ± 0.000	0.0093 ± 0.000	0.0096 ± 0.000
Guava	0.0303 ± 0.016	0.0387 ± 0.006	0.0681 ± 0.000	0.0437 ± 0.002	0.0425 ± 0.005
IOF/ROL	0.7037 ± 0.019	0.6834 ± 0.014	0.7764 ± 0.000	0.7110 ± 0.003	0.7193 ± 0.003
LexisNexis	0.9894 ± 0.005	0.9902 ± 0.006	0.9999 ± 0.000	0.9959 ± 0.000	0.9961 ± 0.000
Paint Control	0.1283 ± 0.000	0.1142 ± 0.001	0.1290 ± 0.000	0.1222 ± 0.000	0.1223 ± 0.000
OkHttp	0.1118 ± 0.014	0.1060 ± 0.003	0.1671 ± 0.000	0.1431 ± 0.000	0.1486 ± 0.000
Retrofit	0.0134 ± 0.001	0.0138 ± 0.001	0.0188 ± 0.000	0.0156 ± 0.000	0.0172 ± 0.000
ZXing	0.0201 ± 0.000	0.0201 ± 0.001	0.0227 ± 0.000	0.0109 ± 0.000	0.0110 ± 0.000
TIME BUDGET: 80%					
Druid	0.1477 ± 0.113	0.1230 ± 0.096	0.4331 ± 0.000	0.4069 ± 0.014	0.4292 ± 0.004
Fastjson	0.0385 ± 0.011	0.0516 ± 0.018	0.1445 ± 0.000	0.1040 ± 0.010	0.0860 ± 0.022
Deeplearning4j	0.5016 ± 0.058	0.5475 ± 0.007	0.5806 ± 0.000	0.4224 ± 0.001	0.4047 ± 0.000
DSpace	0.0374 ± 0.001	0.0269 ± 0.002	0.0606 ± 0.000	0.0525 ± 0.003	0.0526 ± 0.005
GSDTSR	0.0218 ± 0.001	0.0203 ± 0.000	0.0280 ± 0.000	0.0093 ± 0.000	0.0096 ± 0.000
Guava	0.0247 ± 0.013	0.0348 ± 0.009	0.0681 ± 0.000	0.0501 ± 0.012	0.0556 ± 0.011
IOF/ROL	0.7263 ± 0.015	0.6857 ± 0.011	0.7789 ± 0.000	0.7293 ± 0.002	0.7334 ± 0.001
LexisNexis	0.9913 ± 0.005	0.9902 ± 0.006	0.9999 ± 0.000	0.9959 ± 0.000	0.9961 ± 0.000
Paint Control	0.1285 ± 0.000	0.1161 ± 0.001	0.1310 ± 0.000	0.1209 ± 0.000	0.1204 ± 0.000
OkHttp	0.1112 ± 0.017	0.1153 ± 0.003	0.1674 ± 0.000	0.1493 ± 0.000	0.1551 ± 0.000
Retrofit	0.0140 ± 0.001	0.0145 ± 0.001	0.0195 ± 0.000	0.0161 ± 0.000	0.0179 ± 0.000
ZXing	0.0201 ± 0.000	0.0197 ± 0.002	0.0227 ± 0.000	0.0224 ± 0.000	0.0227 ± 0.000

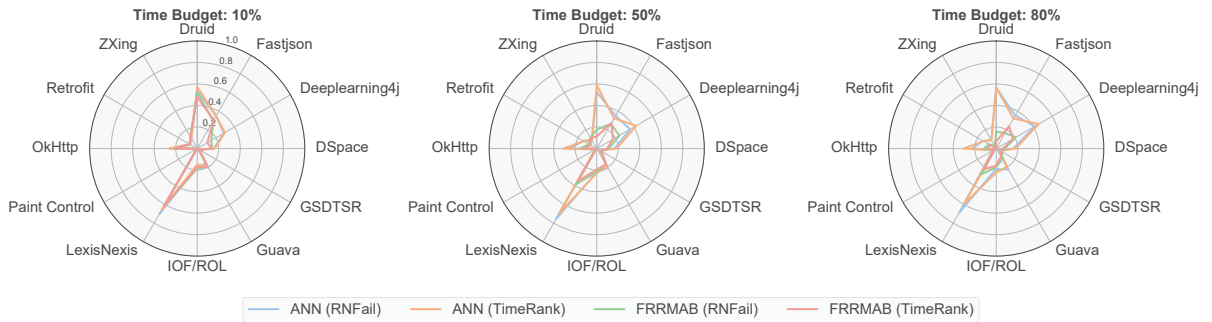


Figure 6: Radar charts - RMSE values found using NAPFD.

Table 7: Prioritization Time (sec.) comparison.

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
TIME BUDGET: 10%					
Druid	0.3035 ± 0.031 ▲	0.3113 ± 0.026 ▲	0.0029 ± 0.000 ★	0.2383 ± 0.005 ▲	0.2316 ± 0.004 ▲
Fastjson	0.2108 ± 0.022 ▲	0.2025 ± 0.012 ▲	0.0029 ± 0.000 ★	0.4947 ± 0.193 ▲	0.4806 ± 0.191 ▲
Deeplearning4j	0.0037 ± 0.000 ▲	0.0179 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0272 ± 0.000 ▲	0.0272 ± 0.000 ▲
DSpace	0.0103 ± 0.001 ▲	0.0305 ± 0.002 ▲	0.0027 ± 0.000 ★	0.0296 ± 0.002 ▲	0.0296 ± 0.002 ▲
GSDTSR	0.0026 ± 0.000 ▲	0.0027 ± 0.000 ▲	0.0020 ± 0.000 ★	0.0253 ± 0.000 ▲	0.0253 ± 0.000 ▲
Guava	0.0335 ± 0.003 ▲	0.0385 ± 0.003 ▲	0.0027 ± 0.000 ★	0.0660 ± 0.013 ▲	0.0660 ± 0.013 ▲
IOF/ROL	0.0031 ± 0.000 ▲	0.2022 ± 0.036 ▲	0.0024 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
LexisNexis	0.4077 ± 0.036 ▲	0.5966 ± 0.052 ▲	0.0037 ± 0.000 ★	0.3328 ± 0.004 ▲	0.3277 ± 0.004 ▲
Paint Control	0.0019 ± 0.000 ★	0.0022 ± 0.000 ▲	0.0022 ± 0.000 ▲	0.0256 ± 0.000 ▲	0.0256 ± 0.000 ▲
OkHttp	0.0074 ± 0.001 ▲	0.0225 ± 0.002 ▲	0.0026 ± 0.000 ★	0.0286 ± 0.001 ▲	0.0285 ± 0.001 ▲
Retrofit	0.0044 ± 0.001 ▲	0.01 ± 0.002 ▲	0.0026 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
ZXing	0.0125 ± 0.001 ▲	0.0151 ± 0.001 ▲	0.0026 ± 0.000 ★	0.0343 ± 0.000 ▲	0.0342 ± 0.000 ▲
TIME BUDGET: 50%					
Druid	0.3881 ± 0.042 ▲	0.3844 ± 0.045 ▲	0.0029 ± 0.000 ★	0.2474 ± 0.004 ▲	0.2373 ± 0.002 ▲
Fastjson	0.2474 ± 0.016 ▲	0.2395 ± 0.028 ▲	0.0029 ± 0.000 ★	0.4774 ± 0.181 ▲	0.4740 ± 0.181 ▲
Deeplearning4j	0.0038 ± 0.000 ▲	0.0187 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0271 ± 0.000 ▲	0.0271 ± 0.000 ▲
DSpace	0.0101 ± 0.001 ▲	0.0507 ± 0.002 ▲	0.0027 ± 0.000 ★	0.0291 ± 0.001 ▲	0.0291 ± 0.001 ▲
GSDTSR	0.0027 ± 0.000 ▲	0.0028 ± 0.000 ▲	0.0021 ± 0.000 ★	0.0253 ± 0.000 ▲	0.0253 ± 0.000 ▲
Guava	0.0335 ± 0.003 ▲	0.0405 ± 0.003 ▲	0.0028 ± 0.000 ★	0.0648 ± 0.012 ▲	0.0648 ± 0.012 ▲
IOF/ROL	0.0034 ± 0.000 ▲	0.5364 ± 0.088 ▲	0.0024 ± 0.000 ★	0.0278 ± 0.000 ▲	0.0278 ± 0.000 ▲
LexisNexis	0.7408 ± 0.114 ▲	1.5037 ± 0.290 ▲	0.0037 ± 0.000 ★	0.3745 ± 0.008 ▲	0.3710 ± 0.008 ▲
Paint Control	0.0020 ± 0.000 ★	0.0028 ± 0.000 ▲	0.0021 ± 0.000 ▲	0.0256 ± 0.000 ▲	0.0256 ± 0.000 ▲
OkHttp	0.0079 ± 0.001 ▲	0.0371 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0283 ± 0.000 ▲	0.0283 ± 0.000 ▲
Retrofit	0.005 ± 0.001 ▲	0.0178 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
ZXing	0.0156 ± 0.002 ▲	0.0229 ± 0.006 ▲	0.0026 ± 0.000 ★	0.0343 ± 0.000 ▲	0.0343 ± 0.000 ▲
TIME BUDGET: 80%					
Druid	0.3733 ± 0.022 ▲	0.2954 ± 0.092 ▲	0.0029 ± 0.000 ★	0.2518 ± 0.003 ▲	0.2393 ± 0.002 ▲
Fastjson	0.2587 ± 0.019 ▲	0.223 ± 0.032 ▲	0.0028 ± 0.000 ★	0.4928 ± 0.185 ▲	0.4795 ± 0.182 ▲
Deeplearning4j	0.0037 ± 0.001 ▲	0.0217 ± 0.008 ▲	0.0025 ± 0.000 ★	0.0272 ± 0.000 ▲	0.0272 ± 0.000 ▲
DSpace	0.0107 ± 0.001 ▲	0.0556 ± 0.001 ▲	0.0025 ± 0.000 ★	0.0293 ± 0.001 ▲	0.0293 ± 0.001 ▲
GSDTSR	0.0026 ± 0.000 ▲	0.0028 ± 0.000 ▲	0.0021 ± 0.000 ★	0.0253 ± 0.000 ▲	0.0253 ± 0.000 ▲
Guava	0.0349 ± 0.003 ▲	0.0405 ± 0.003 ▲	0.0028 ± 0.000 ★	0.0655 ± 0.012 ▲	0.0649 ± 0.012 ▲
IOF/ROL	0.0033 ± 0.001 ▲	0.6649 ± 0.094 ▲	0.0023 ± 0.000 ★	0.0279 ± 0.000 ▲	0.0279 ± 0.000 ▲
LexisNexis	1.0343 ± 0.451 ▲	3.8491 ± 1.681 ▲	0.0036 ± 0.000 ★	0.4130 ± 0.013 ▲	0.4090 ± 0.013 ▲
Paint Control	0.0019 ± 0.000 ★	0.0032 ± 0.001 ▲	0.0021 ± 0.000 ▲	0.0257 ± 0.000 ▲	0.0257 ± 0.000 ▲
OkHttp	0.0076 ± 0.001 ▲	0.0417 ± 0.002 ▲	0.0026 ± 0.000 ★	0.0284 ± 0.000 ▲	0.0284 ± 0.000 ▲
Retrofit	0.0052 ± 0.001 ▲	0.0215 ± 0.004 ▲	0.0025 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
ZXing	0.0188 ± 0.007 ▲	0.0254 ± 0.011 ▲	0.0026 ± 0.000 ★	0.0343 ± 0.000 ▲	0.0343 ± 0.000 ▲

Finding 9. A high test case volatility combined with an increasing number of failures may be a limitation for *RETECS*.

On the other hand, considering the RMSE values for APFDc metric (Table 9), we observe that *RETECS* has better performance than *COLEMAN* in a restrictive scenario, in the other time budgets *COLEMAN* improves, been competitive in time budget of 50% and better than *RETECS* in time budget of 80%. In overall, *COLEMAN* obtained the best results using *TimeRank* function in 17 cases (47%), and *RETECS* using *RNFail* function in 9 cases (25%). Figure 8 shows the radar plot for RMSE values considering APFDc metric.

We observe similar charts to the ones obtained using the NAPFD metric, including a bad performance in the same systems. However, the RMSE values for APFDc are small, that is, both approaches provide good performance to reduce testing costs.

5.6 Answering our RQ

Based on the aforementioned observations, we aim at answering our RQ by defining a scale of RMSE magnitude to represent how far the solutions found by the learning approaches are from the optimal solutions, as follows:

Table 8: RMSE comparison - NAPFD.

SUT	RETECS		COLEMAN	
	RNFail	TimeRank	RNFail	TimeRank
TIME BUDGET: 10%				
Druid	0.5326 ± 0.1449 ▲	0.5716 ± 0.0749 ▲	0.5225 ± 0.0790 ▲	0.4774 ± 0.1162 ▲
Fastjson	0.3513 ± 0.0256 ▲	0.3512 ± 0.0067 ▲	0.2975 ± 0.0261 ▽	0.3080 ± 0.0333 △
Deeplearning4j	0.2858 ± 0.0818 ▽	0.2971 ± 0.0197 ▽	0.1512 ± 0.0031 ▽	0.1077 ± 0.0009 ★
DSpace	0.1529 ± 0.0014 ▽	0.1635 ± 0.0012 ▽	0.1397 ± 0.0056 ★	0.1381 ± 0.0102 ★
GSDTSR	0.0203 ± 0.0002 ★	0.0201 ± 0.0002 ★	0.0190 ± 0.0005 ★	0.0186 ± 0.0005 ★
Guava	0.1583 ± 0.0396 ▽	0.1919 ± 0.0105 ▽	0.1981 ± 0.0030 ▽	0.1930 ± 0.0022 ▽
IOF/ROL	0.1695 ± 0.0116 ▽	0.1476 ± 0.0083 ★	0.1852 ± 0.0027 ▽	0.1759 ± 0.0030 ▽
LexisNexis	0.7026 ± 0.0142 ▲	0.6583 ± 0.0599 ▲	0.6216 ± 0.0017 ▲	0.6175 ± 0.0016 ▲
Paint Control	0.0005 ± 0.0006 ★	0.0026 ± 0.0004 ★	0.0048 ± 0.0001 ★	0.0048 ± 0.0001 ★
OkHttp	0.2126 ± 0.0033 ▽	0.2613 ± 0.0117 ▽	0.2200 ± 0.0000 ▽	0.2064 ± 0.0000 ▽
Retrofit	0.0730 ± 0.0056 ★	0.0835 ± 0.0079 ★	0.0802 ± 0.0000 ★	0.0790 ± 0.0000 ★
ZXing	0.1100 ± 0.0022 ★	0.1097 ± 0.0026 ★	0.1283 ± 0.0000 ★	0.1276 ± 0.0000 ★
TIME BUDGET: 50%				
Druid	0.5264 ± 0.1502 ▲	0.5885 ± 0.0778 ▲	0.1766 ± 0.0357 ▽	0.1129 ± 0.0327 ★
Fastjson	0.3516 ± 0.0105 ▲	0.3234 ± 0.0211 △	0.2645 ± 0.0488 ▽	0.2739 ± 0.0623 ▽
Deeplearning4j	0.3577 ± 0.0773 ▲	0.4235 ± 0.0213 ▲	0.2424 ± 0.0014 ▽	0.1743 ± 0.0006 ▽
DSpace	0.1655 ± 0.0022 ▽	0.1857 ± 0.0026 ▽	0.1116 ± 0.0292 ★	0.0959 ± 0.0271 ★
GSDTSR	0.0282 ± 0.0005 ★	0.0349 ± 0.0003 ★	0.0494 ± 0.0002 ★	0.0492 ± 0.0002 ★
Guava	0.2100 ± 0.0405 ▽	0.1951 ± 0.0114 ▽	0.1704 ± 0.0083 ▽	0.1717 ± 0.0170 ▽
IOF/ROL	0.2183 ± 0.0162 ▽	0.2258 ± 0.0129 ▽	0.2168 ± 0.0041 ▽	0.1937 ± 0.0040 ▽
LexisNexis	0.7598 ± 0.0230 ▲	0.6948 ± 0.0916 ▲	0.3820 ± 0.0074 ▲	0.3543 ± 0.0077 ▲
Paint Control	0.0071 ± 0.0005 ★	0.0160 ± 0.0010 ★	0.0049 ± 0.0004 ★	0.0048 ± 0.0003 ★
OkHttp	0.2531 ± 0.0169 ▽	0.3086 ± 0.0047 △	0.1537 ± 0.0004 ▽	0.1278 ± 0.0006 ★
Retrofit	0.1434 ± 0.0079 ★	0.1384 ± 0.0081 ★	0.0836 ± 0.0000 ★	0.0644 ± 0.0000 ★
ZXing	0.0914 ± 0.0008 ★	0.0897 ± 0.0043 ★	0.1158 ± 0.0000 ★	0.1114 ± 0.0000 ★
TIME BUDGET: 80%				
Druid	0.5656 ± 0.1268 ▲	0.5667 ± 0.1099 ▲	0.1562 ± 0.0312 ▽	0.0666 ± 0.0182 ★
Fastjson	0.3540 ± 0.0105 ▲	0.3221 ± 0.0160 △	0.1656 ± 0.0281 ▽	0.2392 ± 0.0739 ▽
Deeplearning4j	0.4147 ± 0.0949 ▲	0.4567 ± 0.0145 ▲	0.2111 ± 0.0031 ▽	0.1805 ± 0.0013 ▽
DSpace	0.1581 ± 0.0028 ▽	0.1807 ± 0.0031 ▽	0.0803 ± 0.0189 ★	0.0609 ± 0.0271 ★
GSDTSR	0.0312 ± 0.0015 ★	0.0398 ± 0.0003 ★	0.0597 ± 0.0002 ★	0.0595 ± 0.0002 ★
Guava	0.2268 ± 0.0309 ▽	0.1968 ± 0.0190 ▽	0.1131 ± 0.0498 ★	0.0876 ± 0.0572 ★
IOF/ROL	0.1893 ± 0.0155 ▽	0.2226 ± 0.0152 ▽	0.1733 ± 0.0040 ▽	0.1585 ± 0.0033 ▽
LexisNexis	0.6857 ± 0.0433 ▲	0.5898 ± 0.1237 ▲	0.2773 ± 0.0049 ▽	0.2230 ± 0.0037 ▽
Paint Control	0.0360 ± 0.0013 ★	0.0330 ± 0.0022 ★	0.0240 ± 0.0006 ★	0.0236 ± 0.0007 ★
OkHttp	0.2219 ± 0.0260 ▽	0.3007 ± 0.0067 △	0.1131 ± 0.0004 ★	0.0795 ± 0.0007 ★
Retrofit	0.1437 ± 0.0161 ★	0.1362 ± 0.0120 ★	0.0808 ± 0.0000 ★	0.0652 ± 0.0000 ★
ZXing	0.0912 ± 0.0007 ★	0.0892 ± 0.0035 ★	0.0283 ± 0.0000 ★	0.0019 ± 0.0000 ★

$$\text{RMSE Magnitude} = \begin{cases} \text{very near} & \text{if } RMSE < 0.15 \\ \text{near} & \text{if } 0.15 \leq RMSE < 0.23 \\ \text{reasonable} & \text{if } 0.23 \leq RMSE < 0.30 \\ \text{far} & \text{if } 0.30 \leq RMSE < 0.35 \\ \text{very far} & \text{if } 0.35 \leq RMSE \end{cases} \quad (7)$$

where i) the *very near* category (“★”) represents an approximated optimal performance; ii) the *near* category (“▼”) represents reaching optimal performance, and that some improvements are required; iii) the *reasonable* category (“▽”) represents the minimum acceptable performance. Solutions in this category are acceptable and are related to the cases in which the SUT behavior, or possibly the constraints, can make the TCP task hard; iv) the *far* category (“△”) represents unsatisfactory performance, and that meaningful

improvements are required; and v) the *very far* category (“▲”) includes solutions that are far away from to be useful and considered reasonable. By analogy and to a better visualization, we represent the RMSE magnitude with the same symbols used to represent the effect size magnitude in Tables 8 and 9. In this way, we generate Table 10 that presents the distribution of each magnitude for the NAPFD and APFDc values found by COLEMAN and RETECS.

In this path, we consider an approach that finds reasonable solutions when $RMSE < 0.3$. Moreover, other measures suggest such an affirmation. For instance, with 10% of the available time to execute the test cases and considering RMSE for NAPFD values, COLEMAN obtains reasonable solutions with *RNFail* function in 10 out of 12 cases, while with the budgets of 50% and 80% obtains, respectively 11 and 12 cases. However, on the other hand, RETECS using *RNFail*

Table 9: RMSE comparison - APFDc.

SUT	RETECS		COLEMAN	
	RNFail	TimeRank	RNFail	TimeRank
TIME BUDGET: 10%				
Druid	0.3798 ± 0.1186 ▲	0.4221 ± 0.0579 ▲	0.5072 ± 0.0922 ▲	0.4750 ± 0.1176 ▲
Fastjson	0.3340 ± 0.0244 △	0.3343 ± 0.0054 △	0.2935 ± 0.0246 ▼	0.3079 ± 0.0334 △
Deeplearning4j	0.0930 ± 0.0483 ★	0.0966 ± 0.0407 ★	0.1922 ± 0.0036 ▼	0.1839 ± 0.0014 ▼
DSpace	0.1471 ± 0.0012 ★	0.1567 ± 0.0013 ▼	0.1353 ± 0.0044 ★	0.1353 ± 0.0100 ★
GSDTSR	0.0193 ± 0.0002 ★	0.0188 ± 0.0002 ★	0.0191 ± 0.0005 ★	0.0186 ± 0.0005 ★
Guava	0.1203 ± 0.0237 ★	0.1380 ± 0.0105 ★	0.1977 ± 0.0024 ▼	0.1942 ± 0.0019 ▼
IOF/ROL	0.1696 ± 0.0116 ▼	0.1478 ± 0.0083 ★	0.1862 ± 0.0027 ▼	0.1768 ± 0.0030 ▼
LexisNexis	0.7056 ± 0.0140 ▲	0.6625 ± 0.0583 ▲	0.6284 ± 0.0017 ▲	0.6243 ± 0.0016 ▲
Paint Control	0.0005 ± 0.0007 ★	0.0033 ± 0.0006 ★	0.0060 ± 0.0001 ★	0.0060 ± 0.0001 ★
OkHttp	0.1868 ± 0.0028 ▼	0.2273 ± 0.0120 ▼	0.2083 ± 0.0000 ▼	0.1982 ± 0.0000 ▼
Retrofit	0.0636 ± 0.0058 ★	0.0744 ± 0.0084 ★	0.0816 ± 0.0000 ★	0.0803 ± 0.0000 ★
Zxing	0.0940 ± 0.0031 ★	0.0937 ± 0.0035 ★	0.1256 ± 0.0000 ★	0.1243 ± 0.0000 ★
TIME BUDGET: 50%				
Druid	0.3365 ± 0.1361 △	0.4203 ± 0.0602 ▲	0.1650 ± 0.0437 ▼	0.1050 ± 0.0341 ★
Fastjson	0.2018 ± 0.0097 ▼	0.1956 ± 0.0446 ▼	0.2594 ± 0.0492 ▼	0.2691 ± 0.0602 ▼
Deeplearning4j	0.2061 ± 0.0591 ▼	0.1983 ± 0.0248 ▼	0.2513 ± 0.0029 ▼	0.2363 ± 0.0011 ▼
DSpace	0.1401 ± 0.0028 ★	0.1568 ± 0.0033 ▼	0.1085 ± 0.0259 ★	0.0969 ± 0.0252 ★
GSDTSR	0.0267 ± 0.0004 ★	0.0290 ± 0.0003 ★	0.0473 ± 0.0002 ★	0.0471 ± 0.0002 ★
Guava	0.1114 ± 0.0253 ★	0.1049 ± 0.0199 ★	0.1606 ± 0.0084 ▼	0.1708 ± 0.0184 ▼
IOF/ROL	0.2186 ± 0.0166 ▼	0.2315 ± 0.0122 ▼	0.2209 ± 0.0041 ▼	0.1981 ± 0.0038 ▼
LexisNexis	0.7327 ± 0.0172 ▲	0.6780 ± 0.0868 ▲	0.3893 ± 0.0073 ▲	0.3636 ± 0.0075 ▲
Paint Control	0.0059 ± 0.0006 ★	0.0315 ± 0.0019 ★	0.0126 ± 0.0003 ★	0.0126 ± 0.0003 ★
OkHttp	0.2164 ± 0.0290 ▼	0.2114 ± 0.0040 ▼	0.1360 ± 0.0004 ★	0.1192 ± 0.0005 ★
Retrofit	0.1138 ± 0.0086 ★	0.1069 ± 0.0098 ★	0.0731 ± 0.0000 ★	0.0572 ± 0.0000 ★
Zxing	0.0420 ± 0.0018 ★	0.0407 ± 0.0071 ★	0.1078 ± 0.0000 ★	0.1059 ± 0.0000 ★
TIME BUDGET: 80%				
Druid	0.4928 ± 0.1520 ▲	0.5337 ± 0.1015 ▲	0.1562 ± 0.0396 ▼	0.0538 ± 0.0218 ★
Fastjson	0.2859 ± 0.0157 ▼	0.2739 ± 0.0318 ▼	0.1679 ± 0.0268 ▼	0.2309 ± 0.0689 ▼
Deeplearning4j	0.2728 ± 0.0838 ▼	0.2104 ± 0.0238 ▼	0.2842 ± 0.0034 ▼	0.3058 ± 0.0014 △
DSpace	0.1169 ± 0.0030 ★	0.1454 ± 0.0038 ★	0.0782 ± 0.0171 ★	0.0707 ± 0.0260 ★
GSDTSR	0.0284 ± 0.0006 ★	0.0305 ± 0.0003 ★	0.0541 ± 0.0002 ★	0.0539 ± 0.0002 ★
Guava	0.1686 ± 0.0333 ▼	0.1596 ± 0.0281 ▼	0.1134 ± 0.0524 ★	0.0922 ± 0.0584 ★
IOF/ROL	0.1870 ± 0.0160 ▼	0.2291 ± 0.0137 ▼	0.1797 ± 0.0037 ▼	0.1652 ± 0.0032 ▼
LexisNexis	0.6459 ± 0.0365 ▲	0.5812 ± 0.1008 ▲	0.2890 ± 0.0045 ▼	0.2424 ± 0.0037 ▼
Paint Control	0.0321 ± 0.0011 ★	0.0431 ± 0.0025 ★	0.0301 ± 0.0005 ★	0.0295 ± 0.0006 ★
OkHttp	0.2246 ± 0.0408 ▼	0.1846 ± 0.0074 ▼	0.1058 ± 0.0005 ★	0.0791 ± 0.0007 ★
Retrofit	0.1084 ± 0.0152 ★	0.0985 ± 0.0136 ★	0.0702 ± 0.0000 ★	0.0544 ± 0.0000 ★
Zxing	0.0431 ± 0.0014 ★	0.0437 ± 0.0116 ★	0.0258 ± 0.0000 ★	0.0030 ± 0.0000 ★

function obtains reasonable solutions in 9 out of 12 cases with the budget of 10%, and in 8 cases with the budgets of 50% and 80%.

Considering RMSE values for APFDc metric, *COLEMAN* using *RNFail* function, finds reasonable solutions in 10 cases, out of 12, for a time budget of 10%, and 11 and 12 cases for, respectively, the budgets of 50% and 80%. While *RETECS* using *RNFail* finds reasonable solutions in 9 out of 12 cases with the budget of 10%, and in 10 cases for the budgets of 50% and 80%. In such cases, we can conclude that the approaches have, in overall, a good performance.

Considering all 72 cases - all systems, budgets and both measures NAPFD and APFDc - *COLEMAN* obtained reasonable solutions in 66 cases (92%) using *RNFail*, and using *TimeRank* in 63 (88%). On the other hand, *RETECS* obtained reasonable solutions in 54 cases (75%) using *RNFail* and using *TimeRank* in 52 (72%). In overall,

RNFail produced more reasonable solutions than *TimeRank* for both approaches. However, *COLEMAN* using *TimeRank* obtained the best performance obtaining the best RMSE values (highlight in gray in Tables 8 and 9) in 24 cases for NAPFD and in 17 for APFDc, followed by *RETECS* using *RNFail* that obtained 5 cases for NAPFD and 9 for APFDc.

Finding 10. Based on the proposed scale, *COLEMAN* finds reasonable solutions in 92% of the cases and *RETECS* in 75%. The less restrictive the budget the greater the number of reasonable solutions found by both approaches. We can then conclude that the solutions generated are very close to the optimal ones.

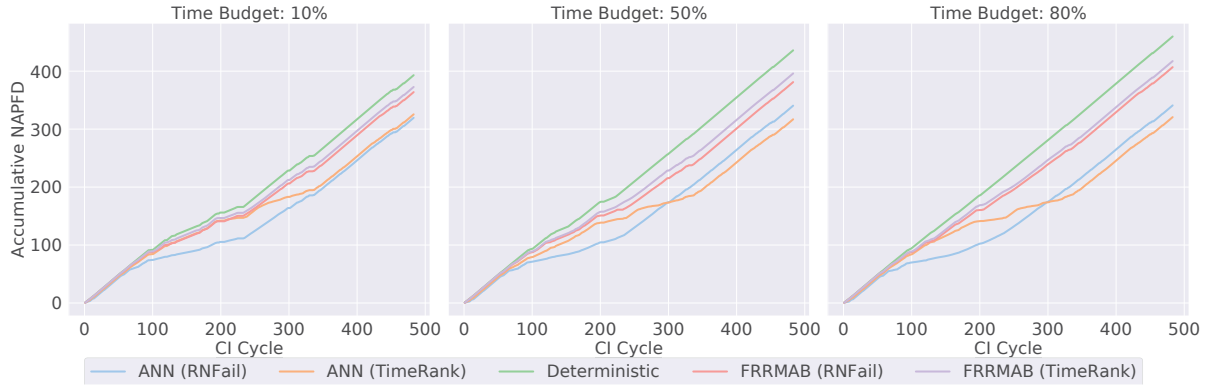


Figure 7: Accumulative NAPFD values for Deeplearning4j system.

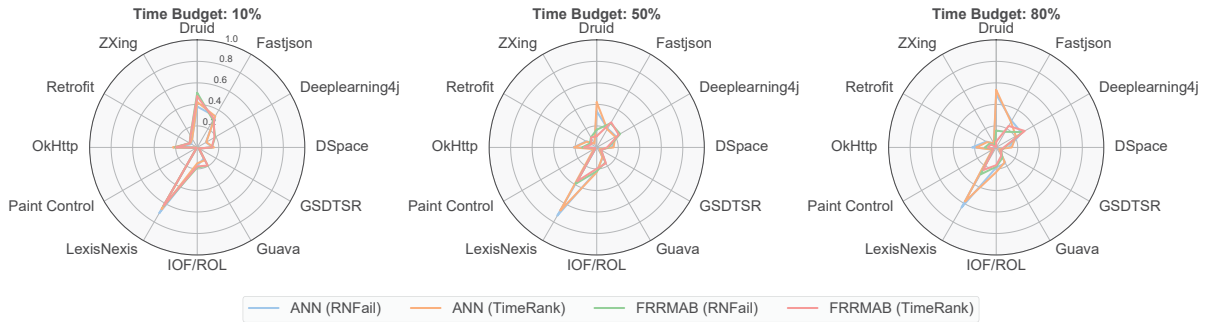


Figure 8: Radar charts - RMSE values found using APFDc.

5.7 Discussions and Implications

In this section, we discuss some implications of our findings regarding the application and limitations of the approaches.

Guidelines for application. Both approaches generate reasonable solutions compared with optimal ones. *COLEMAN* and *RETECS* are able to obtain reasonable solutions in, respectively, 92% and 75% of the cases. They are applicable in real scenarios, spending around one second in the worst case to execute. In this way, they contribute to reducing costs by decreasing the time spent in the CI cycle. *RETECS* presents better performance with *RNFail* function. This happens with all measures evaluated. In contrast with *COLEMAN*, which performs better with *TimeRank*.

COLEMAN outperforms *RETECS* in the great majority of cases. It is important to highlight that this happens for the systems that can be considered hard cases, and considering all measures. However, the use of *RETECS* is indicated in a restrictive budget regarding early fault detection with cost consideration, in this case, test case execution. However, overall considering this cost does not seem to impact the results, nor the performance of *COLEMAN*, which does not include time in its formulation.

Limitations and Improvements. We observe that both approaches have some limitations to learn with few historical test data. The following characteristics are drawbacks for the learning approaches: systems with a small number of CI Cycles, with peaks of failures, and a large test case set, in which many failures are

distributed over many test cases. In this way, a possible research direction is to propose a hybrid approach. An algorithm with good performance with few historical data could be used to overcome that limitation in the first commits. After with enough information *RETECS* or *COLEMAN* would be used. Another possible improvement is the use of Long Short Term Memory (LSTM) networks [10]. The LSTM is well suited to classify, process, and predict time series with time intervals of unknown duration. Gap length insensitivity gives LSTM an advantage over traditional ANNs (used by *RETECS*).

Benchmark. Our analysis revealed some interesting characteristics of the target systems that could be considered in the composition of a benchmark for future experiments. The IOF/ROL, LexisNexis, Deeplearning4j, and Druid systems can be considered the most challenging prioritization cases, due to the high volatility presented, as well as the number of test cases, peaks of failures, and the high number of failures distributed over many test cases. The Guava, Retrofit, and ZXing systems represent scenarios for that it is challenging to obtain expressive time reduction. The failure distribution over the test cases is low and presents small number of peaks in a few CI Cycles. In addition to this, the failing test cases vary in each CI Cycle.

6 THREATS TO VALIDITY

We identified the following points that can be threats to our results. The first threat is the measures used. Other TCP measures could

Table 10: RMSE magnitudes for NAPFD and APFDc values found by *COLEMAN* and *RETECS*.

Scale	NAPFD				APFDc			
	RETECS		COLEMAN		RETECS		COLEMAN	
	RNFail	TimeRank	RNFail	TimeRank	RNFail	TimeRank	RNFail	TimeRank
TIME BUDGET: 10%								
★ <i>very near</i>	4 (33%)	5 (42%)	5 (42%)	6 (50%)	7 (58%)	7 (58%)	5 (42%)	5 (42%)
▼ <i>near</i>	4 (33%)	2 (17%)	4 (33%)	3 (25%)	2 (17%)	2 (17%)	4 (33%)	4 (33%)
▽ <i>reasonable</i>	1 (8%)	2 (17%)	1 (8%)	0 (0%)	0 (0%)	0 (0%)	1 (8%)	0 (0%)
△ <i>far</i>	0 (0%)	0 (0%)	0 (0%)	1 (8%)	1 (8%)	1 (8%)	0 (0%)	1 (8%)
▲ <i>very far</i>	3 (25%)	3 (25%)	2 (17%)	2 (17%)	2 (17%)	2 (17%)	2 (17%)	2 (17%)
TIME BUDGET: 50%								
★ <i>very near</i>	4 (33%)	4 (33%)	5 (42%)	7 (58%)	6 (50%)	5 (42%)	6 (50%)	7 (58%)
▼ <i>near</i>	3 (25%)	3 (25%)	4 (33%)	3 (25%)	4 (33%)	4 (33%)	3 (25%)	2 (17%)
▽ <i>reasonable</i>	1 (8%)	0 (0%)	2 (17%)	1 (8%)	0 (0%)	1 (8%)	2 (17%)	2 (17%)
△ <i>far</i>	0 (0%)	2 (17%)	0 (0%)	0 (0%)	1 (8%)	0 (0%)	0 (0%)	0 (0%)
▲ <i>very far</i>	4 (33%)	3 (25%)	1 (8%)	1 (8%)	1 (8%)	2 (17%)	1 (8%)	1 (8%)
TIME BUDGET: 80%								
★ <i>very near</i>	4 (33%)	4 (33%)	7 (58%)	8 (67%)	5 (42%)	5 (42%)	7 (58%)	8 (67%)
▼ <i>near</i>	4 (33%)	3 (25%)	4 (33%)	3 (25%)	3 (25%)	4 (33%)	3 (25%)	1 (8%)
▽ <i>reasonable</i>	0 (0%)	0 (0%)	1 (8%)	1 (8%)	2 (17%)	1 (8%)	2 (17%)	2 (17%)
△ <i>far</i>	0 (0%)	2 (17%)	0 (8%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (8%)
▲ <i>very far</i>	4 (33%)	3 (25%)	0 (0%)	0 (0%)	2 (17%)	2 (17%)	0 (0%)	0 (0%)
Total of Reasonable Solutions¹	25 (69%)	23 (64%)	33 (92%)	32 (89%)	29 (81%)	29 (81%)	33 (92%)	31 (86%)

¹The reasonable solutions are that ones with RMSE < 0.3.

lead to different results. To mitigate this threat, we chose distinct measures largely used in the TCP literature that better deal with the time budgets and allow us to analyze different perspectives.

The set of parameters used for the approaches is a threat. It is possible that using an automatic configuration setting, the results can get improvements. To minimize such a threat, we used parameters from previous experiments [21, 27] reported in the literature.

The datasets used can be considered a threat. For this, we used a relevant set of systems with different behaviors and aspects concerning the number of failures and test cases.

The last threat is concerning the RMSE magnitude scale. Such magnitudes were obtained based on our analysis, observations, SUTs behavior, and by making correlation with NAPFD and APFDc values. Other researchers can observe different aspects and propose a different scale.

7 CONCLUDING REMARKS

In this paper, we evaluate how far the solutions obtained by TCPCI Ranking-to-Learn approaches, *RETECS* and *COLEMAN*, are from optimal solutions produced by a deterministic approach. We analyzed three test budgets and two reward functions: Reward Based on Failures and Reward Based on Time-Rank, concerning twelve

large-scale real-world software systems. Six measures are used to evaluate: fault detection capability (and cost consideration), early fault detection, time reduction percentage in the CI cycles, prioritization time, and distance from the approximated solution.

Regarding the application of the approaches, *RETECS* reaches the best performance with *RNFail* function, in a less restrictive budget of 10%, and APFDc considering test duration as cost. *COLEMAN* reaches the best performance with *TimeRank* function and mainly for budgets of 50% and 80%. Overall, *COLEMAN* outperforms *RETECS* in the great majority of the cases, considering all systems, budgets, and measures.

Regarding our RQ, we can conclude that both approaches are applicable in real scenarios, taking a negligible time to execute and reducing the CI cycle's time cost. Considering all cases - all systems, budgets and both measures NAPFD and APFDc - *COLEMAN* and *RETECS* produce solutions that are close to the optimal ones in, respectively, 92% and 75% of the cases.

We observe that a high test case volatility, i.e., test case addition or removing along with the CI Cycles, and a high number of failures distributed over many test cases make the problem hard for both approaches. Other findings are that a few cycles can hamper the learning process and that the reduction time in a CI cycle also depends on the test case duration.

Future work includes the use of other evaluation measures to evaluate the approaches. Other systems should be used with a greater number of failures and test cases to allow scalability evaluation.

ACKNOWLEDGMENTS

The work is supported by the Brazilian funding agencies CAPES and CNPq (Grant 305968/2018).

REFERENCES

- [1] Bajaj, A. and Sangwan, O. P. (2019). A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms. *IEEE Access*, 7:126355–126375.
- [2] Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The goal question metric approach. *Encyclopedia of software engineering*, 2(1994):528–532.
- [3] Bertolino, A., Guerriero, A., Breno Miranda, R. P., and Russo, S. (2020). Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *42nd International Conference on Software Engineering, ICSE'20*, pages 1–12, New York, NY, USA. ACM.
- [4] Busjaeger, B. and Xie, T. (2016). Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, pages 975–980, New York, NY, USA. ACM.
- [5] Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2018). A Test Case Prioritization Genetic Algorithm guided by the Hypervolume Indicator. *IEEE Transactions on Software Engineering*.
- [6] Elbaum, S., Malishevsky, A., and Rothermel, G. (2001). Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338.
- [7] Elbaum, S., McLaughlin, A., and Penix, J. (2014). The Google Dataset of Testing Results.
- [8] Epitropakis, M., Yoo, S., Harman, M., and Burke, E. (2015). Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA*, pages 234–245, New York, NY, USA. ACM.
- [9] Haghighathkhan, A., Mäntylä, M., Oivo, M., and Kuvaja, P. (2018). Test prioritization in continuous integration environments. *Journal of Systems and Software*, 146:80–98.
- [10] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.
- [11] Khatibsyarabini, M., Isa, M. A., Jawawi, D. N. A., and Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93.
- [12] Kruskal, W. H. and Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621.
- [13] Kuleshov, V. and Precup, D. (2014). Algorithms for multi-armed bandit problems. *Journal of Machine Learning Research*, 1:1–48.
- [14] Li, K., Fialho, A., Kwong, S., and Zhang, Q. (2014). Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on*, 18(1):114–130.
- [15] Li, Z., Harman, M., and Hierons, R. M. (2007). Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237.
- [16] Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.
- [17] Marijan, D. (2015). Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS'15*, pages 157–162, Washington, DC, USA. IEEE Computer Society.
- [18] Marijan, D., Gotlieb, A., and Liaaen, M. (2019). A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience*, 49(2):192–213.
- [19] Marijan, D., Gotlieb, A., and Sen, S. (2013). Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *IEEE International Conference on Software Maintenance*, pages 540–543. IEEE.
- [20] Marijan, D., Liaaen, M., Gotlieb, A., Sen, S., and Ieva, C. (2017). TITAN: Test Suite Optimization for Highly Configurable Software. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, ICST*, pages 524–531. IEEE.
- [21] Prado Lima, J. A. and Vergilio, S. R. (2020a). A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, page 12.
- [22] Prado Lima, J. A. and Vergilio, S. R. (2020b). Multi-armed bandit test case prioritization in continuous integration environments: A trade-off analysis. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, pages 21–30, New York, NY, USA. Association for Computing Machinery.
- [23] Prado Lima, J. A. and Vergilio, S. R. (2020c). Test Case Prioritization in Continuous Integration Environments: A Systematic Mapping Study. *Information and Software Technology*.
- [24] Qu, X., Cohen, M. B., and Woolf, K. M. (2007). Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *IEEE International Conference on Software Maintenance*, pages 255–264.
- [25] Robbins, H. (1985). Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*, pages 169–177. Springer.
- [26] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (1999). Test Case Prioritization: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99*, pages 179–188. IEEE Computer Society.
- [27] Spieker, H., Gotlieb, A., Marijan, D., and Mossige, M. (2017). Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017*, pages 12–22, New York, NY, USA. ACM.
- [28] Vargha, A. and Delaney, H. D. (2000). A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.
- [29] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.
- [30] Xiao, L., Miao, H., and Zhong, Y. (2018). Test case prioritization and selection technique in continuous integration development environments: a case study. *International Journal of Engineering & Technology*, 7(2.28):332–336.
- [31] Yoo, S. and Harman, M. (2012). Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification & Reliability*, 22(2):67–120.
- [32] Yu, Z., Fahid, F., Menzies, T., Rothermel, G., Patrick, K., and Cherian, S. (2019). TERMINATOR: better automated UI test case prioritization. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, FSE*, pages 883–894. ACM.

APPENDIX E – LEARNING-BASED PRIORITIZATION OF TEST CASES IN CI OF HCS

Learning-based Prioritization of Test Cases in Continuous Integration of Highly-Configurable Software

Jackson A. Prado Lima
DInf, Federal University of
Paraná
Curitiba, Brazil
jackson.lima@ufpr.br

Willian D. F.
Mendonça
DInf, Federal University of
Paraná
Curitiba, Brazil
willian.mendonca@ufpr.br

Silvia R. Vergilio
DInf, Federal University of
Paraná
Curitiba, Brazil
silvia@inf.ufpr.br

Wesley K. G. Assunção
COTSI, Federal University
of Technology - Paraná
Toledo, Brazil
wesleyk@utfpr.edu.br

ABSTRACT

Continuous Integration (CI) is a practice widely adopted in the industry to allow frequent integration of code changes. During the CI process, many test cases are executed multiple times a day, subject to time constraints. In this scenario, a learning-based approach, named COLEMAN, has been successfully applied. COLEMAN allows earlier execution of the most promising test cases to reveal faults. This approach considers CI particularities such as time budget and volatility of test cases, related to the fact that test cases can be added/removed along the CI cycles. In the CI of Highly Configuration System (HCS), many product variants must be tested, each one with different configuration options, but having test cases that are common to or reused from other variants. In this context, we found, by analogy, another particularity, the volatility of variants, that is, some variants can be included/discontinued along CI cycles. Considering this context, this work introduces two strategies for the application of COLEMAN in the CI of HCS: the Variant Test Set Strategy (VTS) that relies on the test set specific for each variant, and the Whole Test Set Strategy (WTS) that prioritizes the test set composed by the union of the test cases of all variants. Both strategies are evaluated in a real-world HCS, considering three test budgets. The results show that the proposed strategies are applicable regarding the time spent for prioritization. They perform similarly regarding early fault detection, but WTS better mitigates the problem of beginning without knowledge, and is more suitable when a new variant to be tested is added.

KEYWORDS

Test Case Prioritization, Family of Products, Software Product Line, Continuous Integration

1 INTRODUCTION

Software systems have to be designed to fulfill demands and requirements of users and customers. However, by nature, users/customers have different needs, mostly related to their business domain, organizational process, environment restriction, and specialized hardware devices. To succeed, software systems must take into account and operate over these different needs. Software Product Line (SPL) is an approach to develop and manage family of software products that can be customized with different configurations (variabilities), while common software assets can be reused in a systematic and disciplined way [15]. SPLs are usually implemented as Highly-Configurable Systems (HCS) by using different configuration options – applying strategies such as conditional compilation,

conditional execution, or build systems – to create custom system products, a.k.a. variants [25, 39].

Modern software development greatly relies on the automation of almost all software engineering processes. For instance, practices like Continuous Integration (CI) have become popular to automatically integrate, build, and test software projects, created by different developers/teams collaboratively [36]. CI leads the software to be integrated and tested multiple times a day to detect integration errors as quickly as possible [6]. However, running a large set of test cases can take many minutes or even hours [33]. For HCSs, when many product variants must be tested, this becomes more problematic. In addition to this, continuous regression testing of HCS is a challenging activity [8]. Some studies on industrial HCS practice conclude that automated tools are needed [8, 41]. In CI, it is fundamental to perform regression testing in a very cost-effective way, providing rapid test feedback on software failures.

Another important point to consider is that, in a company, multiple projects may share the same CI environment, imposing time testing constraints [5]. The test must run a restricted slot of time, referred as *test budget*. Therefore, some traditional regression testing approaches are not suitable for CI, mainly those for selection and minimization of test cases that rely on code coverage, code instrumentation, or search-based techniques that take a long time to execute. In this sense, Test Case Prioritization (TCP) is more popular and used. TCP techniques improve the cost-effectiveness of regression testing by ordering test cases to allow early execution of the most important ones, generally those test cases with high probability of revealing faults [7]. Considering the constraints of test budgets, early fault detection is essential, because when a test case fails, test execution can be ended, and less resources are spent.

A recent mapping reports approaches that adapt TCP techniques for CI environments [30], some of them addressing the HCS context [19, 21, 22]. But most existing approaches present some limitations. Some of them require code analysis, what can be costly. Furthermore, the great majority does not address the main characteristics of CI environments. For instance, they do not consider test case volatility, characteristic associated to the fact that test cases may be added or removed (discontinued) over the CI cycles. They are not adaptive, that is, they do not learn with past prioritizations.

To overcome such limitations, learning approaches based on historical failure data have been proposed. These approaches learn with past prioritizations, usually guided by a reward function [3]. The challenge is to search for early fault detection in the failure-history of past test cases, but also to explore new test cases. This is related to the Exploration versus Exploitation dilemma, and is

a consequence of the test budget, since whether only error-prone test cases are considered without diversity, some test cases can never be executed. COLEMAN (*Combinatorial VOLatiLE Multi-Armed BANDit*) [28] is a promising approach to deal with the Exploration versus Exploitation dilemma. COLEMAN formulates TCP in CI as a Multi-Armed Bandit (MAB) problem [32]. A test case is as an arm and at each time/build multiple arms are selected. In addition, the arms available at each time may change dynamically over time. In this way, it learns with the feedback from the application of the test cases, incorporating diversity in the test suite prioritization.

In the HCS context, we have another particularity that is, by analogy, the volatility of variants that have different configuration options. Each variant can be seen as a system to be individually tested, however, having test cases which are common to or reused from other variants. Also, some variants can be included or discontinued over the CI cycles. To mitigate this problem, this work proposes two strategies for the application of learning-based approaches, such as COLEMAN, in the CI of HCS: (i) the Variant Test Set Strategy (VTS) that relies on the test set specific for each variant; and (ii) the Whole Test Set Strategy (WTS) that prioritizes the test set composed by the union of the test cases of all variants.

VTS and WTS are evaluated in a real-world HCS, namely libssh, regarding some indicators of early fault detection and time reduction. The results allow (i) comparison on the use of COLEMAN with a baseline approach that prioritizes the test cases by randomness; (ii) applicability analysis regarding the time spent in the prioritization and CI cycles; and (iii) comparison of both strategies. The application of the strategies with the learning-based approach COLEMAN produces better results that their application with a random approach. COLEMAN applied with both strategies requires only few seconds to run, not interfering the CI cycles and demonstrating practical usage. Finally, in our evaluation, VTS and WTS present very similar results. However WTS clearly benefits from the cases where there is a history of failed test cases reused among variants, and better mitigates the problem of beginning without knowledge.

The main contribution of this work is the introduction of two strategies for the application of a TCP learning-based approach in CI of HCS. These strategies allow mitigation of the variant volatility problem. VTS and WTS are evaluated using COLEMAN, a MAB based approach that learns from the failure history of reused test cases along the CI cycles, combining exploration and exploitation. The proposed strategies neither require code analysis nor any other model, such as feature model. Moreover, we make public all the data used in this work and the implementation for mining the CI information, which allows replication and future research [27].

The paper is organized as follows. Section 2 presents a motivating example. Section 3 reviews COLEMAN, the learning-based approach adopted in this work. Section 4 introduces the strategies proposed for the HCS context. Section 5 describes how the evaluation was conducted. Section 6 presents and analyzes the results. Section 7 overviews related work. Section 8 presents concluding remarks.

2 MOTIVATING EXAMPLE

To illustrate the importance of using TCP in CI, let us consider the Pipeline #116809311¹ from libssh, the subject system used in

¹<https://gitlab.com/libssh/libssh-mirror/pipelines/116809311/builds>

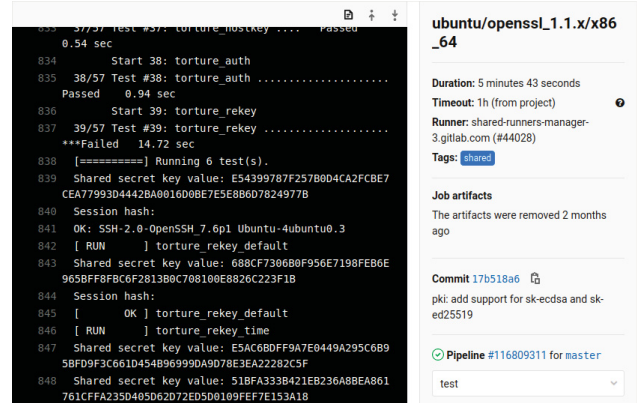


Figure 1: Test #39 failed for ubuntu/openssl_1.1.x/x86_64 after 122.63 seconds of testing.

our evaluation (see Section 5.2). This pipeline aims to perform the integration of a new feature in the system, included in the commit 17b518a6² and described below:

pki: add support for sk-ecdsa and sk-ed25519

This adds server-side support for the newly introduced OpenSSH keytypes sk-ecdsa-sha2-nistp256@openssh.com and sk-ed25519@openssh.com (including their corresponding certificates), which are backed by U2F/FIDO2 tokens.

The aforementioned pipeline contains 27 jobs, each job is responsible for building and testing a different variant of libssh. From these jobs, two of them failed: ubuntu/openssl_1.1.x/x86_64 and visualstudio/x86_64. For instance, Figure 1 presents a screenshot of the build log from variant ubuntu/openssl_1.1.x/x86_64. We can see that the test case #39 (on line 837) has failed.

Considering the testing order defined by the developers/testers, this job took 122.63 seconds until executing the test case that failed. In total, this variant has 57 test cases, and it takes 197.01 seconds to run all tests. Based on that, the testing activity consumed 62% of the runtime to reach the failed test case. In this case, if we had the test budget to 50% of the total runtime and no prioritization, we could not find the bug related to the inclusion of the new feature.

3 COLEMAN

The last section presented the importance of test case orders. Nevertheless, as we mentioned before, TCP in CI for HCSs involves challenging particularities such as time constraints (test budgets), the volatility of test cases, and the volatility of variants. To address such challenges, learning approaches based on historical failure have been proposed. Bertolino et al [3] distinguish two kinds of TCP learning-based approaches. The first one, named Learning-to-rank uses supervised learning to train a model, based on some test features, which is used to rank test sets in future commits. The problem with them is that the model may no longer be representative, when the commit context changes. The second kind, named

²<https://gitlab.com/libssh/libssh-mirror/-/commit/17b518a677c92d943cf016b81272ec10ee1ca368>

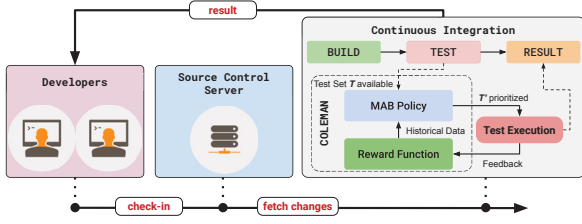


Figure 2: Overview of the COLEMAN interaction with the CI environment (extracted from [29]).

Ranking-to-learn, is more suitable to the dynamic CI context. This strategy learns based on the rewards obtained from the feedback of previous used ranks. The main idea is to maximize the rewards. Ranking-to-learn approaches are more robust regarding the volatility of the test cases, code changes, and number of failing tests. Because of this, the focus of our work is on this kind of approach.

Our study adopts COLEMAN, which presented promising results compared to similar approaches [28]. COLEMAN formulates the TCP in CI as a Multi-Armed Bandit (MAB) problem. MAB problems are sequential decision problems related to the scenario in which a player plays on a set of slot machines (or arms/actions) that even being identical produce different gains [32]. After a player pulls one of the arms in a turn, a reward is received from some unknown distribution, aiming to maximize the sum of the rewards. To this end, a policy is the strategy that chooses, at each time, the next arm to pull based on previously observed rewards and decisions.

To represent the TCP in CI problem as a MAB problem, COLEMAN considers that a test case is an arm to be pulled, and it encompasses two MAB variations, namely combinatorial and volatile, to deal with the dynamic nature of our problem.

In the former variation, at each turn (commit/build/CI Cycle), a MAB policy selects all arms (test cases) available instead of one. According to the order in which the policy selects the arms, the prioritization is being defined. The latter variation deal with the test case volatility. In this case, only the test set available at each commit is used by the policy.

COLEMAN is generic and lightweight approach and requires only historical failure data. No further detail about the system under test is required, such as code coverage or code instrumentation. Figure 2 shows how COLEMAN interacts with the CI environment. In such an environment, teams work continuously integrating code and making smaller code commits every day, usually monitored by a CI server. When a change occurs, the CI server clones this code, builds it and runs the testing processes (on the right side of the figure). When the entire process is finished, a report is generated by the CI server, and the developers are informed. COLEMAN acts after a successful build, in the test phase the approach prioritizes the test case set available to be used during the test case execution.

For each build (commit c), a test case set T_c is available, as well as a test budget. When the budget is smaller than the total time required to execute T_c , our approach is then used to obtain a prioritized test set T'_c . The idea is that the most relevant test cases, i.e., the ones that fail, are executed first. Then, according to the performance of its previous prioritization, the policy receives a reward (feedback).

Based on rewards provided by a reward function, the policy adapts its experience for future actions (*online learning*). COLEMAN can be used with different MAB policies and reward functions. But as presented and evaluated in previous studies [28], COLEMAN obtained the best performance using TimeRank (*Time-Ranked*) function and FRRMAB (*Fitness-Rate-Rank based on Multi-Armed Bandit*) policy [14]. For this reason, we used TimeRank and FRRMAB in this study. They are described as follows.

The *TimeRank* function is defined in Equation 1. This function is based on the rank of t'_c in $T'_c \forall t'_c \in T'_c$, where T'^{fail} is composed by the failing test cases from T'_c ; $RNFail$ returns 1 if t'_c failed and 0 otherwise. The $prec(t'_{c_1}, t'_{c_2})$ function returns 1 if the position of t'_{c_1} is lower than the position of t'_{c_2} . The idea is to evaluate whether failing test cases are ranked in the first positions in T'_c . To this end, a test case t'_c that does not fail and precedes failing ones are penalized by their early scheduling. A non-failed test case receives a reward given by the accumulated number of test cases which failed until its position in the prioritization rank, that is, it receives a reward decreased by the number of failing test cases ranked after it.

$$TimeRank(t'_c) = |T'^{fail}| - [\neg(RNFail(t'_c)) \times \sum_{i=1}^{|T'^{fail}|} prec(t'_c, t'_{c_i})] \quad (1)$$

FRRMAB policy works with a sliding window with size W . The reward value (FIR for FRRMAB), is obtained through the reward function. In this way, it is considered the last W commits as historical test data. Thus, for each test case, FRRMAB policy considers the history of rewards whilst other policies use cumulative rewards. During the execution, if a test case is discontinued in a commit (build), it is then removed along with its history.

4 APPLICATION STRATEGIES

COLEMAN was initially designed to deal with a traditional CI process and its particularities. However, the HCS context also has particularities, namely variant volatility. Next we introduce two strategies to deal with variant volatility in the CI of HCS.

For the integration of a commit c of an HCS S , COLEMAN was adapted to consider that there is a set V of n variants. For each variant $v_i \in V$ a set of test cases $T_{v_i,c}$ is available. That is, in this study we consider that there is only one test set by variant. COLEMAN receives as input n test sets, $\{T_{v_1,c}, T_{v_2,c}, \dots, T_{v_n,c}\}$, and produces n prioritized test sets, $\{T'_{v_1,c}, T'_{v_2,c}, \dots, T'_{v_n,c}\}$. The value n can vary depending on the number of existing variants of S when c is committed. For the COLEMAN prioritization, we propose two strategies to deal with a set of variants and volatility, as illustrated in Figure 3.

1. Variant Test Set Strategy (VTS): in this strategy (at the top of Figure 3) COLEMAN is applied n times for each c treating each variant independently. The i^{th} application has as input the set T_{v_i,c_i} and as output the prioritized set T'_{v_i,c_i} . When a new variant of S is introduced, no history information is available.

2. Whole Test Set Strategy (WTS): for this strategy (at the bottom of Figure 3) COLEMAN is applied only once for each c and has as input only one test set $TS_c = \bigcup_{i=1}^n T_{v_i,c}$ composed by the union of all test sets of all variants under test. A test case t in TS_c can be common/reused to more than one variant, but it appears in TS only

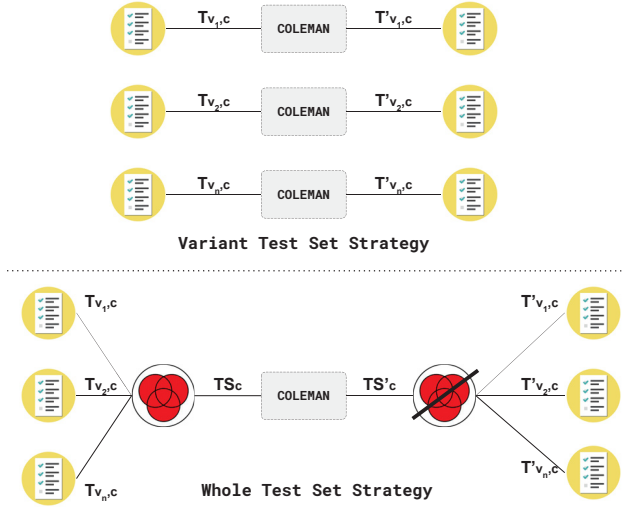


Figure 3: COLEMAN strategies to deal with HCS variants and their volatility.

once. The status of t is set to failed whether t failed in at least one variant in which the test case have been executed previously. To calculate the total time required to execute TS we set the duration of t with the maximum execution time for t considering all variants. The output is a prioritized set TS'_c . The sets $T'_{v_i,c}$ are then generated, by selecting from TS'_c only the same test cases that belong to $T_{v_i,c}$ but keeping the prioritization order.

The main advantage of WTS is that if a new variant appears, it can be tested based on the historical information collected from the other ones. In this way, mitigating the problem of beginning without knowledge (learning) and adapting to changes in the execution environment, either by test case volatility or variant volatility.

5 EVALUATION SETUP

This section describes details of our study conducted to evaluate COLEMAN. The goal is to evaluate the applicability of the proposed strategies and their performance in comparison with a baseline random approach. A replication package is available in [27].

5.1 Research Questions

The following research questions guide the evaluation:

- RQ1:** *How is the performance of both strategies when a random approach is used in comparison with COLEMAN?* This question compares the results obtained by VTS and WTS strategies using COLEMAN with a prioritization order generated randomly, approach commonly used in the industry.
- RQ2:** *Using COLEMAN, are the strategies applicable for HCSs in the CI development context?* This question investigates whether the time spent in the prioritization is acceptable considering CI Cycles (commits).
- RQ3:** *What is the best strategy in the HCS context?* This question aims to compare the proposed strategies, VTS and WTS.

The idea is to evaluate their ability regarding early fault detection, and percentage of reduced time.

5.2 Subject System

Our evaluation is based on the SSH library (libssh)³, which is an open-source C multiplatform library implementing the SSHv2 protocol on client and server side. This library is designed to remotely execute programs, transfer files, use a secure and transparent tunnel, manage public keys, and the like. libssh is statically configurable with the C preprocessor. libssh was used in the literature by other pieces work on the topic of HCS and SPLs [10, 23, 24, 26].

Libssh is hosted on GitLab⁴, which provides an environment with control version system and CI pipelines. The logs of the CI pipeline jobs are the source of information for COLEMAN. Figure 4 presents a real example of log from a job⁵ of a libssh. The log describes information related to the configuration, build, and execution of test cases of a variant. This latter is the basis to collect input information for COLEMAN. We can observe in Figure 4 that the execution of test cases started on line 759, and the result of the first test case is seen on line 761. This line 761 has the three pieces of information that are collected: (1) The name of the test case, in this example “torture_buffer”; (2) the status of the execution, which in this case is “Passed”, but can also be “Failed”, as we can see in the bottom of the figure, on line 837; and (3) execution time that in this test case on 761 is equal to “0.26 sec”. We make available⁶ the tool to collect such pieces of information.

The libssh mirror on GitLab was mined on 2020-04-02, from where we collected the information of Table 1. We can find 44 product variants of libssh on GitLab, however, we selected 31 that have at least one failed test case in their history of builds. The first line of the table presents information of the set composed by the union of the test cases related to all variants, which is considered to evaluate WTS. The first two columns shows the variant name and an acronym, used to improve the visualization in the analysis. The third column shows the period that each variant was used. The fourth column presents the total of builds identified. Only valid builds (success or fail) were considered, that is, we discarded builds with some problem, for instance, canceled builds. The fifth column shows the total of failures found, and in parentheses the number of builds in which at least one test failed. The sixth column shows the number of different (unique) test cases identified from build logs, and in parentheses the range of test cases executed in the builds. The last columns present the mean (\pm standard deviation) duration in minutes of the CI Cycles and the interval between them.

5.3 Quality Indicators

We adopted indicators from TCP literature [28, 31]: the Normalized Average Percentage of Fault Detected (NAPFD), the Rank of the Failing Test Cases (RFTC), Normalized Time Reduction (NTR), and the Prioritization Time (PT). They are defined as follows.

NAPFD [31] is an extension of the Average Percentage of Faults Detected (APFD) [34]. APFD measures how fast a set of prioritized

³<https://www.libssh.org/>

⁴<https://gitlab.com/libssh/libssh-mirror>

⁵<https://gitlab.com/libssh/libssh-mirror/-/jobs/432862254#L837>

⁶<https://github.com/jacksonpradolima/gitlabci-torrent>

Table 1: Test Case Set Information

Name	Acronym	Period	Builds	Faults	Tests	Duration (min)	Interval (min)
Total		2018/04/12-2020/02/25	334	223 (127)	62 (17 - 62)	0.4029 (2.037)	1142.528 (459.283)
address-sanitizer	addr-san	2018/04/18-2018/04/18	1	13 (1)	29 (29 - 29)	0.0110 (0.032)	-
CentOS7-openssl	cent-op1	2018/04/12-2018/04/12	1	1 (1)	17 (17 - 17)	0.0237 (0.042)	-
CentOS7-openssl_1.0.x-x86-64	cent-op2	2018/04/12-2018/04/18	22	8 (7)	29 (17 - 29)	0.0104 (0.02)	1320.7984 (287.806)
centos7-openssl_1.0.x-x86-64	cent-op3	2018/04/18-2019/12/10	128	1 (1)	26 (17 - 25)	0.0332 (0.1)	972.3706 (457.855)
Debian.cross.mips-linux-gnu	debi-cross	2018/07/02-2018/12/24	131	7 (7)	29 (19 - 29)	1.3815 (4.122)	956.783 (499.357)
Debian-openssl_1.0.x-aarch64	debi-op	2018/04/12-2018/04/13	8	1 (1)	17 (17 - 17)	0.0431 (0.083)	1298.6738 (306.178)
fedora-libcrypt-x86_64	fed-lib	2018/09/10-2020/02/25	179	11 (11)	57 (40 - 57)	0.0669 (0.204)	930.5167 (533.409)
fedora-mbedtls-x86-64	fed-mb1	2018/06/27-2019/12/10	108	16 (6)	40 (30 - 40)	0.0554 (0.16)	987.9657 (458.517)
fedora-mbedtls-x86_64	fed-mb2	2018/09/10-2020/02/25	178	2 (2)	56 (39 - 56)	0.0568 (0.177)	904.7565 (534.632)
Fedora-openssl	fed-op1	2018/04/12-2018/04/12	1	1 (1)	17 (17 - 17)	0.0149 (0.031)	-
Fedora-openssl_1.1.x-x86-64	fed-op2	2018/04/12-2018/04/18	21	3 (3)	29 (17 - 29)	0.0120 (0.027)	1315.1925 (293.81)
fedora-openssl_1.1.x-x86-64	fed-op3	2018/04/18-2019/12/10	125	14 (13)	41 (29 - 41)	0.0489 (0.132)	993.4781 (455.632)
fedora-openssl_1.1.x-x86-64-release	fed-op-r	2018/08/20-2019/03/13	57	4 (4)	41 (38 - 41)	0.0485 (0.126)	927.5525 (463.556)
fedora-openssl_1.1.x-x86_64	fed-op4	2018/09/10-2020/02/25	179	3 (3)	60 (40 - 60)	0.0604 (0.195)	905.6718 (535.739)
fedora-openssl_1.1.x-x86_64-flips	fed-op-f	2019/06/13-2020/02/25	71	11 (10)	57 (55 - 57)	0.0805 (0.297)	1013.623 (538.089)
fedora-openssl_1.1.x-x86_64-minimal	fed-op-m	2018/12/13-2020/02/25	124	1 (1)	45 (40 - 45)	0.0461 (0.15)	927.5229 (534.695)
fedora-undefined-sanitizer	fed-und	2018/04/18-2020/02/25	307	16 (15)	57 (29 - 57)	0.0701 (0.208)	966.4755 (506.954)
freebsd-x86_64	freebsd	2018/09/10-2020/02/25	179	10 (3)	33 (24 - 33)	0.0346 (0.102)	886.7128 (543.546)
mingw32	mingw32	2018/06/27-2019/12/10	86	3 (3)	16 (10 - 16)	0.0849 (0.229)	951.7082 (482.182)
mingw64	mingw64	2018/06/27-2019/12/10	86	3 (3)	16 (10 - 16)	0.0610 (0.142)	957.3851 (477.583)
pages	pages	2018/04/18-2018/04/18	3	18 (3)	29 (29 - 29)	0.0088 (0.022)	1434.4083 (0.608)
tumbleweed-openssl_1.1.x-x86-64	tumb-op	2018/05/30-2019/12/10	103	24 (8)	41 (29 - 41)	0.0568 (0.15)	963.3945 (482.27)
tumbleweed-openssl_1.1.x-x86-64-release	tumb-op-r	2018/08/20-2019/03/13	57	1 (1)	41 (38 - 41)	0.0600 (0.155)	942.0249 (457.872)
tumbleweed-openssl_1.1.x-x86_64-clang	tumb-op-c	2018/09/10-2020/02/25	180	19 (10)	57 (38 - 57)	0.0616 (0.184)	909.4417 (536.359)
tumbleweed-openssl_1.1.x-x86_64-gcc	tumb-op-g	2018/09/10-2020/02/25	176	16 (7)	56 (38 - 56)	0.0620 (0.184)	896.6659 (536.939)
tumbleweed-openssl_1.1.x-x86_64-gcc7	tumb-op-g7	2018/09/10-2020/02/25	179	15 (7)	57 (38 - 57)	0.0625 (0.187)	902.7045 (540.006)
tumbleweed-undefined-sanitizer	tumb-und	2018/05/30-2020/02/25	300	36 (14)	56 (29 - 56)	0.0707 (0.196)	956.385 (514.161)
ubuntu-openssl_1.1.x-x86_64	ubun-op	2019/12/23-2020/02/25	13	10 (5)	57 (57 - 57)	0.0583 (0.19)	1036.3259 (543.659)
undefined-sanitizer	und-san	2018/04/18-2018/04/18	4	29 (2)	29 (29 - 29)	0.0162 (0.043)	1430.6278 (1.341)
visualstudio-x86	vs-x86	2018/11/30-2020/02/25	116	13 (11)	18 (16 - 18)	0.0044 (0.007)	953.6386 (530.41)
visualstudio-x86_64	vs-x86_64	2018/11/30-2020/02/25	116	14 (14)	18 (16 - 18)	0.0051 (0.01)	957.3534 (534.557)

```

758 [100%] Built target test_server
759 Test project /builds/libssh/libssh-mirror/obj
760 Start 1: torture_buffer
761 1/57 Test #1: torture_buffer ..... Passed 0.26 sec
762 Start 2: torture_bytearray
763 2/57 Test #2: torture_bytearray ..... Passed 0.00 sec
764 Start 3: torture_callbacks
765 3/57 Test #3: torture_callbacks ..... Passed 0.00 sec
766 Start 4: torture_crypto
767 4/57 Test #4: torture_crypto ..... Passed 0.00 sec
768 Start 5: torture_init
...
837 39/57 Test #39: torture_rekey .....***Failed 14.72 sec
838 [*****] Running 6 test(s).
839 Shared secret key value: E54399787F257B0D4CA2FCBE7CEA77993D4442BA0016D0
BE7E5E8B6D7824977B
840 Session hash:
841 OK: SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
842 [ RUN ] torture_rekey_default
843 Shared secret key value: 688CF7306B0F956E7198FEB6E965BFF8FBC6F2813B0C70
8100E8826C223F1B

```

Figure 4: Example of Job Log

test cases (T') can success on detecting faults in the program under tested. APFD values range from zero to one and is computed from the weighted average of the percentage of detected faults. Higher values indicate that the faults are detected faster using fewer test cases. On the other hand, NAPFD metric is adequate for prioritization of test cases when not all of them are executed, and some faults can be undetected. NAPFD, in addition to APFD, considers the ratio between detected and detectable faults within T . Equation 2 describes how to compute NAPFD, where m is the number of faults detected by all test cases; $rank(T'_i)$ is the position of T'_i in T' . If T'_i did not reveal faults, then it is set to $T'_i = 0$. n is the number of tests cases in T' and p is the number of faults detected by T' divided by m .

$$NAPFD(T'_i) = p - \frac{\sum_1^n rank(T'_i)}{m \times n} \frac{p}{2n} \quad (2)$$

RFCT [28] evaluates how fast a prioritized test set T' detects a fault. RFCT considers the i^{th} position from the first test case in T' that fails. In this way, the values range from 1 to $|T'|$, and lower values represent a faster failure detection. This indicator can be used as a base to stop the testing process when reaching failure.

NTR [28] (Equation 3) evaluates the capability to reduce the time spent in a CI Cycle. It measures the difference between the time spent to reach the first failed test case r_t and the time to execute all tests \hat{r}_t . In such a metric, only commits that fail CI^{fail} are considered. The values range from 0 to 1, in which higher values represent a higher test time reduction.

$$NTR(\mathcal{A}) = \frac{\sum_{t=1}^{CI^{fail}} (\hat{r}_t - r_t)}{\sum_{t=1}^{CI^{fail}} (\hat{r}_t)} \quad (3)$$

PT takes into account the runtime spent by an approach to perform the prioritization in each commit. This value is used to provide an indicative of the applicability of the proposed strategies in real scenarios. In our experiments we measure PT in seconds.

5.4 Applying learning and random approaches

To apply the learning-based approach, COLEMAN, we adopted the same settings reported in previous work [28]. We used the same configuration for FRRMAB: sliding window size W equals to 100, the coefficient C to balance exploration and exploitation equals to 0.3, and decayed factor equals to 1. We defined three configurations of test budgets: 10%, 50%, and 80% of the execution time of the overall test set available in each commit. These different test budgets allow us to investigate the behavior of each approach and strategy during the TCP in CI process.

The results were obtained from a total of 5,760 executions performing 30 independent executions for each approach (FRRMAB and Random) in each variant (31 variants), strategy (2 strategies), and time budget (3 time budgets). As mentioned before, in VTS strategy each variant is tested separately, whilst in WTS strategy all variants are executed as a unique system. All the experiments were performed on an Intel® Xeon® E5-2450 with 2.10 GHz CPU, 47GB RAM, running Linux Ubuntu 18.04.1 LTS.

5.5 Statistical Analysis

To evaluate a pair of performances in the same variant, we used Mann-Whitney [18] statistical test with a confidence level of 95%. Furthermore, we used the Vargha and Delaney's \hat{A}_{12} [38] to calculate the difference between two groups. \hat{A}_{12} evaluate the probability of a value, taken randomly from the first sample, is higher than a value taken randomly from the second sample. This metric provides a magnitude scale: (i) *Negligible*, represents a very small difference among the values and usually does not yield statistical difference; (ii) *Small* and *Medium*, represent small and medium differences among the values, and may yield statistical differences; and (iii) *Large* magnitude represents a significantly large difference that usually can be seen in the numbers without much effort.

6 RESULTS

In this section, the results obtained are presented and discussed in order to answer the posed questions.

6.1 RQ1: COLEMAN vs Random

To answer RQ1, we compare the results of both strategies using a random prioritization to the results obtained using COLEMAN with the FRRMAB policy. Such an analysis encompasses the three time budgets. To compare the performance, we used NAPFD as the main quality indicator. Table 2 presents the results obtained. The best values are highlighted in bold, and values that are statistically equivalent to the best ones have their corresponding cells painted in light gray. Different symbols are used to indicate the effect size magnitude concerning the best values (see Table 2 footnote).

As we can observe, COLEMAN using FRRMAB is the best approach with a statistical difference in most cases. Considering 31 variants, both strategies and the three time budgets ($31 \times 2 \times 3 = 186$ cases), COLEMAN is the best approach with statistical difference in 157 cases (84.4%). The random approach is the best only in 8 cases (4.3%) and reaches statistically equivalent results in 21 cases (11.3%). Furthermore, when there is a statistical difference, the effect size tends to be large (\blacktriangle). On the other hand, cases with statistical equivalence appear across the time budgets. In these few cases, the variants have few failures (most of them have only one failure), and clearly, COLEMAN has not significant performance.

We also can see that COLEMAN outperforms the random approach, independently of the strategy used. If we use the VTS strategy, FRRMAB is the best one in 87 out of 93 cases (31 variants \times 3 budgets) and presents equivalent results in 6. If we use WTS, FRRMAB is the best in 70 cases (out of 93). It is statistically equivalent in 15 cases, and is the worst in 8 cases.

The worst performance of the approach is obtained for the variant address-sanitizer (addr-san). In this variant, we have only

one commit and 13 test cases that fail from 29 available. Before this variant, only 8 variants are defined, and 12 commits have some test case which failed. From these commits, the failed test cases are scattered between the variants. From the test set available for this variant, only 9 test cases have historical failure data. From the available test cases, some take more time (duration) to execute than others, and among them, the failed test cases. This shows that sometimes we face test cases that fail but spend much time to execute, and this hampers a reasonable prioritization. Another similar case can be observed in the second worst performance, for the undefined-sanitizer (und-san) variant.

The biggest difference can be observed in the variant ubuntu-openssl_1.1.x-x86_64 (ubun-op). Although this variant has few commits, there is a high test case volatility, mainly in two commits in which failures occur⁷. Before these two commits, only one commit failed.

RQ1: *Independently of the strategy and time budget, we can then conclude COLEMAN outperforms the random approach. It reaches the best results or statistically equivalent ones in the great majority of the cases ($\approx 96\%$).*

6.2 RQ2: Strategies Applicability

To analyze the applicability of the strategies using COLEMAN, we compute PT, which is the prioritization time in seconds, spent by them considering the three budgets. The results are depicted in Table 3. We observe that both strategies spend in all cases less than one second. In the worst case, VTS spends 0.0435 seconds in the variant CentOS7-openssl (cent-op1) and time budget of 10%.

Regarding Table 1, we observe that a commit is typically performed after another one is finished and with a considered time. Considering the commit duration, the variants used do not present a situation with multiple test requests for the same variant. The approach is applicable considering the time between commits.

RQ2: *Regarding the time spent to prioritize the test cases, both strategies, VTS and WTS, spend less than one second to execute. In addition to this, we do not observe any impact of the time budgets. Considering the aforementioned facts, the time spent by the strategies is feasible, that is, both strategies are applicable in the CI context.*

6.3 RQ3: Comparing VTS and WTS strategies

To compare both strategies we analyse the NAPFD values, presented in Table 2. The statistical test results between them are available in the supplementary material [27].

Analyzing such a table, we observe that VTS is better than WTS in time budget of 10%. For this budget VTS obtains the best performance, with statistical difference, in 15 cases (out of 31, $\approx 48\%$). On the other hand, WTS is the best in only $\approx 13\%$ (4 cases). Considering the time budgets of 50% and 80%, VTS is the best in few cases, respectively, $\approx 45\%$ (14) and $\approx 29\%$ (9) cases, whilst WTS is the best in $\approx 32\%$ (10) and $\approx 45\%$ (14). In this sense, WTS provides better results with less restrictive situation (more time budget), and VTS the opposite. Considering all budgets and variants (93 cases),

⁷See supplementary material about characteristics of this variant. Available in [27].

Table 2: NAPFD values comparing COLEMAN using FRRMAB and Random approaches.

Variant	TIME BUDGET 10%				TIME BUDGET 50%				TIME BUDGET 80%			
	VTS		WTS		VTS		WTS		VTS		WTS	
	FRRMAB	Random	FRRMAB	Random	FRRMAB	Random	FRRMAB	Random	FRRMAB	Random	FRRMAB	Random
addr-san	0.7573 ± 0.000 ★	0.5710 ± 0.049 ▲	0.5887 ± 0.049 ★	0.5586 ± 0.050 ▲	0.7663 ± 0.004 ★	0.5127 ± 0.047 ▲	0.5226 ± 0.053	0.5035 ± 0.068	0.7635 ± 0.005 ★	0.4829 ± 0.071 ▲	0.5107 ± 0.068	0.4916 ± 0.053
cent-op1	0.9706 ± 0.000 ★	0.6078 ± 0.245 ▲	0.6902 ± 0.194	0.6000 ± 0.226	0.9706 ± 0.000 ★	0.5922 ± 0.290 ▲	0.6235 ± 0.265	0.5176 ± 0.264	0.9706 ± 0.000 ★	0.5118 ± 0.329 ▲	0.6127 ± 0.300	0.4627 ± 0.331
cent-op2	0.9742 ± 0.012 ★	0.8287 ± 0.034 ▲	0.8517 ± 0.059	0.8231 ± 0.041	0.9828 ± 0.002 ★	0.8400 ± 0.036 ▲	0.9659 ± 0.011 ★	0.8361 ± 0.038 ▲	0.9841 ± 0.003 ★	0.8516 ± 0.046 ▲	0.9706 ± 0.012 ★	0.8487 ± 0.031 ▲
cent-op3	0.9922 ± 0.000	0.9922 ± 0.000	0.9922 ± 0.000	0.9922 ± 0.000	0.9979 ± 0.000 ★	0.9968 ± 0.002 ▲	0.9998 ± 0.000 ★	0.9962 ± 0.002 ▲	0.9975 ± 0.000 ★	0.9946 ± 0.003 ▲	0.9922 ± 0.000	0.9952 ± 0.003 ★
debi-cross	0.9466 ± 0.000	0.9466 ± 0.000	0.9466 ± 0.000	0.9466 ± 0.000	0.9527 ± 0.000	0.9549 ± 0.007	0.9669 ± 0.010 ★	0.9548 ± 0.007 ▲	0.9949 ± 0.000 ★	0.9666 ± 0.006 ▲	0.9979 ± 0.002 ★	0.9674 ± 0.006 ▲
debi-op	0.9963 ± 0.000 ★	0.9556 ± 0.026 ▲	0.9963 ± 0.000 ★	0.9534 ± 0.028 ▲	0.9963 ± 0.000 ★	0.9404 ± 0.030 ▲	0.9963 ± 0.000 ★	0.9534 ± 0.032 ▲	0.9963 ± 0.000 ★	0.9426 ± 0.039 ▲	0.9963 ± 0.000 ★	0.9294 ± 0.033 ▲
fed-11b	0.9385 ± 0.000	0.9385 ± 0.000	0.9385 ± 0.000	0.9385 ± 0.000	0.9988 ± 0.000 ★	0.9563 ± 0.008 ▲	0.9985 ± 0.001 ★	0.9560 ± 0.007 ▲	0.9986 ± 0.000 ★	0.9680 ± 0.006 ▲	0.9979 ± 0.001 ★	0.9660 ± 0.006 ▲
fed-mb1	0.9596 ± 0.000 ★	0.9525 ± 0.002 ▲	0.9596 ± 0.000 ★	0.9535 ± 0.003 ▲	0.9743 ± 0.000 ★	0.9636 ± 0.003 ▲	0.9832 ± 0.000 ★	0.9638 ± 0.004 ▲	0.9946 ± 0.000 ★	0.9663 ± 0.004 ▲	0.9946 ± 0.000 ★	0.9663 ± 0.004 ▲
fed-mb2	0.9935 ± 0.000 ★	0.9922 ± 0.001 ▲	0.9943 ± 0.000 ★	0.9916 ± 0.002 ▲	0.9991 ± 0.000 ★	0.9927 ± 0.003 ▲	0.9965 ± 0.003 ★	0.9946 ± 0.003 ▽	0.9980 ± 0.002 ★	0.9946 ± 0.003 ▲	0.9980 ± 0.002 ★	0.9946 ± 0.003 ▲
fed-op1	0.9706 ± 0.000 ★	0.6078 ± 0.245 ▲	0.5843 ± 0.208	0.5863 ± 0.188	0.9706 ± 0.000 ★	0.6039 ± 0.281 ▲	0.4882 ± 0.213	0.5196 ± 0.249	0.9706 ± 0.000 ★	0.5412 ± 0.291 ▲	0.5059 ± 0.258	0.5588 ± 0.219
fed-op2	0.9932 ± 0.009 ★	0.9411 ± 0.020 ▲	0.9744 ± 0.014 ★	0.9429 ± 0.023 ▲	0.9910 ± 0.006 ★	0.9441 ± 0.029 ▲	0.9764 ± 0.012 ★	0.9310 ± 0.017 ▲	0.9923 ± 0.006 ★	0.9405 ± 0.024 ▲	0.9701 ± 0.011 ★	0.9343 ± 0.023 ▲
fed-op3	0.9722 ± 0.001 ★	0.9236 ± 0.009 ▲	0.9834 ± 0.002 ★	0.9227 ± 0.010 ▲	0.9896 ± 0.000 ★	0.9369 ± 0.012 ▲	0.9895 ± 0.004 ★	0.9374 ± 0.009 ▲	0.9851 ± 0.000 ★	0.9468 ± 0.009 ▲	0.9893 ± 0.004 ★	0.9445 ± 0.008 ▲
fed-op-r	0.9473 ± 0.000 ★	0.9350 ± 0.007 ▲	0.9298 ± 0.000 ▽	0.9335 ± 0.007 ★	0.9986 ± 0.000 ★	0.9552 ± 0.014 ▲	0.9965 ± 0.004 ★	0.9537 ± 0.015 ▲	0.9989 ± 0.000 ★	0.9595 ± 0.011 ▲	0.9916 ± 0.009 ★	0.9631 ± 0.011 ▲
fed-op4	0.9882 ± 0.000 ★	0.9855 ± 0.003 ▲	0.9873 ± 0.001 ★	0.9858 ± 0.003 ▲	0.9972 ± 0.000 ★	0.9899 ± 0.004 ▲	0.9929 ± 0.001 ★	0.9895 ± 0.003 ▲	0.9968 ± 0.000 ★	0.9916 ± 0.003 ▲	0.9961 ± 0.002 ★	0.9900 ± 0.003 ▲
fed-op-f	0.8797 ± 0.000 ★	0.8670 ± 0.010 ▲	0.9257 ± 0.025 ★	0.8700 ± 0.014 ▲	0.9962 ± 0.000 ★	0.9104 ± 0.018 ▲	0.9955 ± 0.003 ★	0.9072 ± 0.015 ▲	0.9964 ± 0.000 ★	0.9199 ± 0.013 ▲	0.9981 ± 0.001 ★	0.9240 ± 0.015 ▲
fed-op-m	0.9919 ± 0.000	0.9919 ± 0.000	0.9919 ± 0.000	0.9919 ± 0.000	0.9992 ± 0.000 ★	0.9954 ± 0.003 ▲	0.9999 ± 0.000 ★	0.9956 ± 0.003 ▲	0.9992 ± 0.000 ★	0.9951 ± 0.003 ▲	0.9999 ± 0.000 ★	0.9958 ± 0.002 ▲
fed-und	0.9714 ± 0.000 ★	0.9606 ± 0.004 ▲	0.9711 ± 0.002 ★	0.9619 ± 0.004 ▲	0.9910 ± 0.000 ★	0.9711 ± 0.005 ▲	0.9963 ± 0.001 ★	0.9701 ± 0.005 ▲	0.9939 ± 0.000 ★	0.9738 ± 0.004 ▲	0.9975 ± 0.001 ★	0.9727 ± 0.004 ▲
freebsd	0.9957 ± 0.000 ★	0.9915 ± 0.002 ▲	0.9995 ± 0.000 ★	0.9921 ± 0.002 ▲	0.9983 ± 0.000 ★	0.9914 ± 0.002 ▲	0.9995 ± 0.000 ★	0.9919 ± 0.002 ▲	0.9983 ± 0.000 ★	0.9912 ± 0.002 ▲	0.9995 ± 0.000 ★	0.9915 ± 0.002 ▲
mingw32	0.9813 ± 0.000 ★	0.9814 ± 0.008 ▲	0.9918 ± 0.004 ★	0.9817 ± 0.007 ▲	0.9901 ± 0.000 ★	0.9843 ± 0.005 ▲	0.9925 ± 0.003 ★	0.9828 ± 0.006 ▲	0.9901 ± 0.000 ★	0.9833 ± 0.005 ▲	0.9930 ± 0.003 ★	0.9836 ± 0.006 ▲
mingw64	0.9913 ± 0.000 ★	0.9808 ± 0.008 ▲	0.9919 ± 0.005 ★	0.9821 ± 0.008 ▲	0.9901 ± 0.000 ★	0.9843 ± 0.005 ▲	0.9928 ± 0.003 ★	0.9842 ± 0.006 ▲	0.9901 ± 0.000 ★	0.9833 ± 0.005 ▲	0.9930 ± 0.003 ★	0.9843 ± 0.005 ▲
pages	0.6022 ± 0.092 ★	0.3609 ± 0.104	0.6065 ± 0.145 ★	0.3495 ± 0.093 ▲	0.8570 ± 0.005 ★	0.4885 ± 0.088 ▲	0.7670 ± 0.044 ★	0.4548 ± 0.072 ▲	0.8543 ± 0.005 ★	0.5066 ± 0.076 ▲	0.7683 ± 0.051 ★	0.4769 ± 0.085 ▲
tumb-op	0.9797 ± 0.000 ★	0.9453 ± 0.009 ▲	0.9752 ± 0.000 ★	0.9435 ± 0.008 ▲	0.9923 ± 0.000 ★	0.9585 ± 0.007 ▲	0.9932 ± 0.000 ★	0.9567 ± 0.007 ▲	0.9902 ± 0.000 ★	0.9611 ± 0.006 ▲	0.9947 ± 0.000 ★	0.9587 ± 0.006 ▲
tumb-op-r	0.9825 ± 0.000	0.9836 ± 0.004	0.9825 ± 0.000 ▽	0.9847 ± 0.006 ★	0.9997 ± 0.000 ★	0.9873 ± 0.006 ▲	0.9969 ± 0.002 ★	0.9880 ± 0.007 ▲	0.9997 ± 0.000 ★	0.9910 ± 0.006 ▲	0.9972 ± 0.003 ★	0.9904 ± 0.005 ▲
tumb-op-c	0.9708 ± 0.000 ★	0.9569 ± 0.004 ▲	0.9574 ± 0.006	0.9573 ± 0.005	0.9961 ± 0.000 ★	0.9669 ± 0.005 ▲	0.9865 ± 0.012 ★	0.9651 ± 0.005 ▲	0.9953 ± 0.000 ★	0.9700 ± 0.004 ▲	0.9947 ± 0.003 ★	0.9708 ± 0.005 ▲
tumb-op-g	0.9823 ± 0.000 ★	0.9692 ± 0.003 ▲	0.9618 ± 0.002 ▲	0.9686 ± 0.003 ★	0.9976 ± 0.000 ★	0.9772 ± 0.004 ▲	0.9639 ± 0.001 ▲	0.9758 ± 0.004 ★	0.9975 ± 0.000 ★	0.9791 ± 0.004 ▲	0.9974 ± 0.001 ★	0.9782 ± 0.004 ▲
tumb-op-g7	0.9772 ± 0.000 ★	0.9695 ± 0.003 ▲	0.9609 ± 0.000 ▲	0.9693 ± 0.003 ★	0.9977 ± 0.000 ★	0.9769 ± 0.004 ▲	0.9992 ± 0.000 ★	0.9775 ± 0.005 ▲	0.9977 ± 0.000 ★	0.9796 ± 0.004 ▲	0.9991 ± 0.000 ★	0.9782 ± 0.003 ▲
tumb-und	0.9821 ± 0.001 ★	0.9672 ± 0.004 ▲	0.9749 ± 0.000 ★	0.9658 ± 0.004 ▲	0.9950 ± 0.000 ★	0.9738 ± 0.004 ▲	0.9750 ± 0.000 ★	0.9738 ± 0.004 ▽	0.9936 ± 0.000 ★	0.9757 ± 0.004 ▲	0.9940 ± 0.000 ★	0.9762 ± 0.002 ▲
ubun-op	0.9443 ± 0.000 ★	0.6754 ± 0.044 ▲	0.9204 ± 0.000 ★	0.6877 ± 0.055 ▲	0.9599 ± 0.000 ★	0.7810 ± 0.057 ▲	0.9204 ± 0.000 ★	0.7533 ± 0.058 ▲	0.9599 ± 0.000 ★	0.7932 ± 0.055 ▲	0.9966 ± 0.000 ★	0.8061 ± 0.056 ▲
und-san	0.8057 ± 0.004 ★	0.7149 ± 0.040 ▲	0.7326 ± 0.037	0.7178 ± 0.042	0.8628 ± 0.001 ★	0.7520 ± 0.031 ▲	0.8294 ± 0.011 ★	0.7479 ± 0.032 ▲	0.8628 ± 0.001 ★	0.7520 ± 0.031 ▲	0.8270 ± 0.010 ★	0.7600 ± 0.016 ▲
vs-x86	0.9792 ± 0.000 ★	0.9185 ± 0.010 ▲	0.9052 ± 0.000 ▲	0.9205 ± 0.010 ★	0.9925 ± 0.000 ★	0.9442 ± 0.013 ▲	0.9841 ± 0.018 ★	0.9401 ± 0.012 ▲	0.9930 ± 0.000 ★	0.9466 ± 0.008 ▲	0.9945 ± 0.005 ★	0.9495 ± 0.011 ▲
vs-x86_64	0.9704 ± 0.000 ★	0.8933 ± 0.012 ▲	0.8793 ± 0.000 ▲	0.9001 ± 0.015 ★	0.9891 ± 0.000 ★	0.9301 ± 0.011 ▲	0.9516 ± 0.034 ★	0.9309 ± 0.011 ▲	0.9891 ± 0.000 ★	0.9316 ± 0.010 ▲	0.9913 ± 0.006 ★	0.9343 ± 0.010 ▲

This table reports the NAPFD results (averages ± standard deviation) obtained from 30 independent runs with three time budgets: 10%, 50%, and 80%; and organized by variant for Variant test set (VTS) and Whole test set strategies (WTS). Values highlighted in bold with a “★” symbol denotes the best approach for a strategy in a variant and, in gray, results that are statistically equal to the best one. A “▼” indicates that the effect size was negligible in relation to the best, while “▽” denotes a small magnitude, “▲” a medium magnitude, and “▲” a large magnitude.

Table 3: PT values considering Variant test set and Whole test set strategies.

Variant	TIME BUDGET 10%		TIME BUDGET 50%		TIME BUDGET 80%	
	VTS	WTS	VTS	WTS	VTS	WTS
addr-san	0.0370 ± 0.001 ★	0.0423 ± 0.003 ▲	0.0368 ± 0.000	0.0390 ± 0.004	0.0370 ± 0.000 ★	0.0424 ± 0.003 ▲
cent-op1	0.0428 ± 0.007	0.0425 ± 0.006	0.0384 ± 0.004	0.0382 ± 0.005	0.0402 ± 0.006 ▲	0.0354 ± 0.001 ★
cent-op2	0.0370 ± 0.001	0.0363 ± 0.001 ★	0.0371 ± 0.001	0.0363 ± 0.001 ★	0.0369 ± 0.001	0.0367 ± 0.001
cent-op3	0.0392 ± 0.000 ▲	0.0372 ± 0.000 ★	0.0392 ± 0.000	0.0370 ± 0.000 ★	0.0393 ± 0.000	0.0372 ± 0.001 ★
debi-cross	0.0405 ± 0.000 ▲	0.0373 ± 0.000 ★	0.0403 ± 0.000	0.0373 ± 0.000 ★	0.0404 ± 0.000	0.0375 ± 0.000 ★
debi-op	0.0361 ± 0.001	0.0361 ± 0.001	0.0368 ± 0.001	0.0365 ± 0.001	0.0362 ± 0.001	0.0361 ± 0.001
fed-11b	0.0422 ± 0.000 ▲	0.0412 ± 0.001 ★	0.0421 ± 0.000	0.0413 ± 0.001 ★	0.0421 ± 0.000	0.0411 ± 0.000 ★
fed-mb1	0.0396 ± 0.000 ▲	0.0391 ± 0.000 ★	0.0395 ± 0.000	0.0390 ± 0.000 ★	0.0396 ± 0.000	0.0387 ± 0.001 ★
fed-mb2	0.0422 ± 0.000 ▲	0.0409 ± 0.000 ★	0.0421 ± 0.000	0.0411 ± 0.001 ★	0.0421 ± 0.000	0.0410 ± 0.000 ★
fed-op1	0.0428 ± 0.007	0.0401 ± 0.006	0.0384 ± 0.004 ★	0.0418 ± 0.004	0.0402 ± 0.006	0.0416 ± 0.005
fed-op2	0.0369 ± 0.001 ▲	0.0364 ± 0.001 ★	0.0370 ± 0.001	0.0376 ± 0.002	0.0368 ± 0.001	0.0366 ± 0.001
fed-op3	0.0392 ± 0.000	0.0391 ± 0.001	0.0392 ± 0.000	0.0393 ± 0.001	0.0392 ± 0.000	0.0390 ± 0.001
fed-op-r	0.0406 ± 0.000 ▲	0.0386 ± 0.000 ★	0.0406 ± 0.000	0.0384 ± 0.000 ★	0.0406 ± 0.000	0.0391 ± 0.001 ★
fed-op4	0.0422 ± 0.000 ▲	0.0411 ± 0.001 ★	0.0421 ± 0.000	0.0412 ± 0.001 ★	0.0421 ± 0.000	0.0411 ± 0.000 ★
fed-op-f	0.0430 ± 0.000 ▲	0.0408 ± 0.001 ★	0.0431 ± 0.000	0.0407 ± 0.001 ★	0.0430 ± 0.000	0.0404 ± 0.000 ★
fed-op-m	0.0426 ± 0.000 ▲	0.0398 ± 0.001 ★	0.0426 ± 0.000	0.0397 ± 0.000 ★	0.0426 ± 0.000	0.0398 ± 0.000 ★
fed-und	0.0410 ± 0.000	0.0409 ± 0.000	0.0409 ± 0.000	0.0411 ± 0.000	0.0409 ± 0.000	0.0413 ± 0.001
freebsd	0.0422 ± 0.000 ▲	0.0380 ± 0.000 ★	0.0421 ± 0.000	0.0381 ± 0.000 ★	0.0421 ± 0.000	0.0381 ± 0.000 ★
mingw32	0.0395 ± 0.000 ▲	0.0360 ± 0.001 ★	0.0395 ± 0.000	0.0361 ± 0.001 ★	0.0395 ± 0.000	0.0361 ± 0.001 ★
mingw64	0.0395 ± 0.000 ▲	0.0360 ± 0.001 ★	0.0395 ± 0.000	0.0367 ± 0.001 ★	0.0395 ± 0.000	0.0359 ± 0.001 ★
pages	0.0375 ± 0.002 ▽	0.0370 ± 0.001 ★	0.0374 ± 0.002	0.0389 ± 0.004	0.0368 ± 0.000	0.0375 ± 0.002
tumb-op	0.0392 ± 0.000 ▲	0.0387 ± 0.001 ★	0.0391 ± 0.000	0.0386 ± 0.001 ★	0.0392 ± 0.000	0.0386 ± 0.000 ★
tumb-op-r	0.0406 ± 0.000 ▲	0.0386 ± 0.000 ★	0.0406 ± 0.000	0.0382 ± 0.000 ★	0.0406 ± 0.000	0.0385 ± 0.001 ★
tumb-op-c	0.0422 ± 0.000 ▲	0.0413 ± 0.000 ★	0.0421 ± 0.000	0.0411 ± 0.000 ★	0.0421 ± 0.000	0.0412 ± 0.000 ★
tumb-op-g	0.0422 ± 0.000 ▲	0.0411 ± 0.000 ★	0.0421 ± 0.000	0.0407 ± 0.000 ★	0.0421 ± 0.000	0.0410 ± 0.000 ★
tumb-op-g7	0.0422 ± 0.000 ▲	0.0410 ± 0.000 ★	0.0421 ± 0.000	0.0408 ± 0.000 ★	0.0421 ± 0.000	0.0411 ± 0.001 ★
tumb-und	0.0411 ± 0.000	0.0410 ± 0.000	0.0410 ± 0.000	0.0408 ± 0.000	0.0410 ± 0.000	0.0408 ± 0.000
ubun-op	0.0434 ± 0.000 ▲	0.0406 ± 0.001 ★	0.0436 ± 0.000	0.0400 ± 0.001 ★	0.0435 ± 0.000	0.0405 ± 0.001 ★
und-san	0.0371 ± 0.001	0.0376 ± 0.001	0.0374 ± 0.002	0.0384 ± 0.003	0.0371 ± 0.001	0.0378 ± 0.002
vs-x86	0.0424 ± 0.000 ▲	0.0365 ± 0.000 ★	0.0424 ± 0.000	0.0363 ± 0.000 ★	0.0424 ± 0.000	0.0362 ± 0.000 ★
vs-x86_64	0.0424 ± 0.000 ▲	0.0364 ± 0.000 ★	0.0424 ± 0.000	0.0363 ± 0.000 ★	0.0424 ± 0.000	0.0366 ± 0.001 ★

Table 4: RFCTC and NTR values considering VTS and WTS strategies.

Variant	TIME BUDGET 10%				TIME BUDGET 50%				TIME BUDGET 80%			
	RFCTC		NTR		RFCTC		NTR		RFCTC		NTR	
	VTS	WTS	VTS	WTS	VTS	WTS	VTS	WTS	VTS	WTS	VTS	WTS
addr-san	1.0000 ± 0.000 ★	1.9000 ± 0.995 ▲	0.9990 ± 0.000	0.9600 ± 0.117	1.0000 ± 0.000 ★	2.0667 ± 1.172 ▲	0.9990 ± 0.000	0.9278 ± 0.154	1.0000 ± 0.000 ★	1.9667 ± 1.273 ▲	0.9990 ± 0.000	0.9167 ± 0.166
cent-op1	1.0000 ± 0.000 ★	5.7667 ± 3.298 ▲	0.9954 ± 0.000	0.6066 ± 0.341	1.0000 ± 0.000 ★	6.9000 ± 4.513 ▲	0.9954 ± 0.000	0.5458 ± 0.403	1.0000 ± 0.000	5.9259 ± 3.892	0.9954 ± 0.000	0.5736 ± 0.334
cent-op2	1.2667 ± 0.270 ★	2.4452 ± 0.801 ▲	0.3252 ± 0.001	0.1948 ± 0.062	1.6286 ± 0.217 ★	2.4286 ± 0.667 ▲	0.3231 ± 0.003	0.3027 ± 0.015	1.4857 ± 0.269 ★	2.1143 ± 0.660 ▲	0.3242 ± 0.003	0.3075 ± 0.011
cent-op3	-	-	0.0000 ± 0.000	0.0000 ± 0.000	6.0000 ± 0.000 ▲	1.0000 ± 0.000 ★	0.0015 ± 0.000	0.0055 ± 0.000	7.0000 ± 0.000	-	0.0015 ± 0.000	0.0000 ± 0.000
debi-cross	-	-	0.0000 ± 0.000	0.0000 ± 0.000	10.0000 ± 0.000	2.0000 ± 3.672	0.0097 ± 0.000	0.0302 ± 0.016	5.4286 ± 0.000 ▲	2.0643 ± 0.738 ★	0.0644 ± 0.000	0.0601 ± 0.006
debi-op	1.0000 ± 0.000	1.0000 ± 0.000	0.1522 ± 0.000	0.1522 ± 0.000	1.0000 ± 0.000	1.0000 ± 0.000	0.1522 ± 0.000	0.1522 ± 0.000	1.0000 ± 0.000	1.0000 ± 0.000	0.1522 ± 0.000	0.1522 ± 0.000
fed-lib	-	-	0.0000 ± 0.000	0.0000 ± 0.000	2.2727 ± 0.000	2.5636 ± 1.480	0.0827 ± 0.000	0.0820 ± 0.002	2.6364 ± 0.000	3.5091 ± 2.038	0.0799 ± 0.000	0.0811 ± 0.002
fed-mb1	1.0000 ± 0.000	1.0000 ± 0.000	0.0213 ± 0.000	0.0213 ± 0.000	1.0000 ± 0.000 ★	1.6444 ± 0.122 ▲	0.0213 ± 0.000	0.0482 ± 0.002	1.6500 ± 0.141	1.6500 ± 0.141	0.0480 ± 0.002	0.0480 ± 0.002
fed-mb2	6.7000 ± 0.466 ▲	1.0000 ± 0.000 ★	0.0035 ± 0.000	0.0041 ± 0.000	4.0000 ± 0.000 ▲	1.6000 ± 0.747 ★	0.0112 ± 0.000	0.0072 ± 0.004	16.2333 ± 15.476	16.2333 ± 15.476	0.0110 ± 0.002	0.0110 ± 0.002
fed-op1	1.0000 ± 0.000 ★	7.5667 ± 3.530 ▲	0.9934 ± 0.000	0.4491 ± 0.400	1.0000 ± 0.000 ★	9.2000 ± 3.624 ▲	0.9934 ± 0.000	0.3798 ± 0.383	1.0000 ± 0.000 ★	8.9000 ± 4.381 ▲	0.9934 ± 0.000	0.4999 ± 0.346
fed-op2	1.3222 ± 0.557 ★	3.3056 ± 1.183 ▲	0.1438 ± 0.011	0.1201 ± 0.016	2.0889 ± 1.184 ★	3.4111 ± 1.515 ▲	0.1441 ± 0.004	0.1248 ± 0.016	1.8222 ± 1.140 ★	4.2556 ± 1.498 ▲	0.1449 ± 0.003	0.1232 ± 0.014
fed-op3	3.5133 ± 0.855 ★	5.0556 ± 0.103 ▲	0.0916 ± 0.000	0.1010 ± 0.003	4.6154 ± 0.000	3.1675 ± 1.613	0.1168 ± 0.000	0.1174 ± 0.004	4.1556 ± 0.029 ▲	2.7449 ± 1.539 ★	0.1050 ± 0.000	0.1147 ± 0.008
fed-op-r	1.0000 ± 0.000	-	0.0241 ± 0.001	0.0000 ± 0.000	2.0000 ± 0.000	4.3500 ± 4.287	0.1020 ± 0.000	0.1017 ± 0.004	1.7500 ± 0.000	9.7667 ± 10.202	0.1052 ± 0.000	0.0995 ± 0.006
fed-op4	9.0000 ± 0.000	21.9667 ± 18.648	0.0052 ± 0.000	0.0053 ± 0.003	18.3333 ± 0.000 ▲	14.8000 ± 8.814 ★	0.0181 ± 0.001	0.0171 ± 0.000	20.6667 ± 0.000	21.9000 ± 9.804	0.0148 ± 0.000	0.0225 ± 0.003
fed-op-f	5.0000 ± 0.000	1.9112 ± 1.365 ★	0.1201 ± 0.042	0.1201 ± 0.042	3.7000 ± 0.000	4.2333 ± 2.367	0.2263 ± 0.000	0.2236 ± 0.003	3.5000 ± 0.000	1.9733 ± 0.481 ★	0.2299 ± 0.000	0.2259 ± 0.002
fed-op-m	-	-	0.0000 ± 0.000	0.0000 ± 0.000	5.0000 ± 0.000 ▲	1.0000 ± 0.000	0.0081 ± 0.000	0.0083 ± 0.000	5.0000 ± 0.000	1.0000 ± 0.000	0.0081 ± 0.000	0.0083 ± 0.000
fed-und	3.6190 ± 0.427 ★	3.9103 ± 0.228 ▲	0.0181 ± 0.001	0.0165 ± 0.003	3.0769 ± 0.000	2.4824 ± 0.501 ★	0.0464 ± 0.000	0.0523 ± 0.001	3.5048 ± 0.090 ▲	2.2713 ± 0.532 ★	0.0497 ± 0.000	0.0528 ± 0.001
freedsb	8.6667 ± 0.000 ▲	1.0000 ± 0.000 ★	0.0281 ± 0.000	0.0282 ± 0.000	5.3333 ± 0.000 ▲	1.0000 ± 0.000 ★	0.0281 ± 0.000	0.0282 ± 0.000	5.3333 ± 0.000 ▲	1.0000 ± 0.000 ★	0.0281 ± 0.000	0.0282 ± 0.000
mingw32	3.0000 ± 0.000	1.4667 ± 0.571 ★	0.0106 ± 0.000	0.0085 ± 0.001	3.3333 ± 0.000 ▲	2.6556 ± 0.908 ★	0.0077 ± 0.000	0.0091 ± 0.002	3.3333 ± 0.000 ▲	2.3000 ± 0.984 ★	0.0077 ± 0.000	0.0091 ± 0.002
mingw64	3.0000 ± 0.000	1.3333 ± 0.438 ★	0.0138 ± 0.000	0.0114 ± 0.002	3.3333 ± 0.000 ▲	2.5556 ± 0.932 ★	0.0099 ± 0.000	0.0120 ± 0.002	3.3333 ± 0.000 ▲	2.5000 ± 0.883 ★	0.0099 ± 0.000	0.0113 ± 0.002
pages	2.1444 ± 0.168	1.7556 ± 0.716 ★	0.9424 ± 0.007	0.9448 ± 0.075	2.0000 ± 0.000	1.9667 ± 0.809	0.9487 ± 0.000	0.9538 ± 0.049	2.0222 ± 0.085 ★	2.1000 ± 1.554 ▲	0.9469 ± 0.005	0.9551 ± 0.062
tumb-op	1.7190 ± 0.026 ▲	1.4286 ± 0.000 ★	0.0331 ± 0.000	0.0301 ± 0.000	1.7500 ± 0.000 ▲	1.5000 ± 0.000 ★	0.0289 ± 0.000	0.0334 ± 0.000	1.7500 ± 0.000 ▲	1.5000 ± 0.000 ★	0.0289 ± 0.000	0.0334 ± 0.000
tumb-op-r	3.5000 ± 0.000	8.7333 ± 10.529	0.0000 ± 0.000	0.0000 ± 0.000	2.0000 ± 0.000 ★	14.4667 ± 11.085 ▲	0.0268 ± 0.000	0.0261 ± 0.001	2.0000 ± 0.000 ★	13.0000 ± 11.185 ▲	0.0268 ± 0.000	0.0262 ± 0.001
tumb-op-c	2.0000 ± 0.000	1.0000 ± 0.000	0.0579 ± 0.000	0.0275 ± 0.020	5.3000 ± 0.000	6.6150 ± 6.346	0.0683 ± 0.000	0.0670 ± 0.020	6.3000 ± 0.000	8.2641 ± 4.901	0.0654 ± 0.000	0.0785 ± 0.006
tumb-op-g	2.1667 ± 0.000	-	0.0601 ± 0.000	0.0046 ± 0.007	3.0000 ± 0.000	3.7667 ± 2.176	0.0471 ± 0.000	0.0128 ± 0.001	3.0000 ± 0.000	1.2667 ± 0.304 ★	0.0471 ± 0.000	0.0597 ± 0.001
tumb-op-g7	-	-	0.0456 ± 0.000	0.0000 ± 0.000	3.4286 ± 0.000 ▲	1.5333 ± 0.507 ★	0.0410 ± 0.000	0.0547 ± 0.001	3.4286 ± 0.000 ▲	1.6333 ± 0.490 ★	0.0410 ± 0.000	0.0545 ± 0.001
tumb-und	2.6265 ± 0.662 ▲	2.4286 ± 0.000 ★	0.0390 ± 0.000	0.0119 ± 0.000	3.8571 ± 0.000 ▲	2.1619 ± 0.290 ★	0.0306 ± 0.000	0.0132 ± 0.000	2.7179 ± 0.037 ▲	1.6923 ± 0.192 ★	0.0302 ± 0.000	0.0399 ± 0.000
ubun-op	17.2000 ± 0.000 ▲	1.1250 ± 0.127 ★	0.5635 ± 0.000	0.5193 ± 0.001	12.6000 ± 0.000	1.1167 ± 0.127 ★	0.3935 ± 0.000	0.5202 ± 0.002	12.6000 ± 0.000	1.1000 ± 0.102 ★	0.3935 ± 0.000	0.5702 ± 0.002
und-san	1.0000 ± 0.000 ★	1.3167 ± 0.425 ▲	0.5217 ± 0.000	0.5196 ± 0.013	1.0000 ± 0.000 ★	1.3167 ± 0.404 ▲	0.5217 ± 0.000	0.5166 ± 0.018	1.0000 ± 0.000 ★	1.3500 ± 0.418 ▲	0.5217 ± 0.000	0.5165 ± 0.018
vs-x86	2.0000 ± 0.000	-	0.1625 ± 0.000	0.0000 ± 0.000	3.4545 ± 0.000	2.0884 ± 1.092 ★	0.2132 ± 0.000	0.1902 ± 0.043	3.3636 ± 0.000 ▲	2.4879 ± 1.822 ★	0.2134 ± 0.000	0.2105 ± 0.018
vs-x86_64	1.8182 ± 0.000	-	0.1604 ± 0.000	0.0000 ± 0.000	3.4286 ± 0.000	1.8373 ± 0.659	0.2078 ± 0.000	0.1261 ± 0.059	3.4286 ± 0.000	2.9024 ± 1.587	0.2078 ± 0.000	0.2032 ± 0.015

(See caption of Table 2 for a description of the headings.)

hand, we use the NTR metric to observe the impact of the order to reduce the test duration process.

Table 4 presents the RFCTC and NTR values. We observe that, considering the RFCTC values, VTS is the best strategy in 12 cases for time budget 10%, 8 for 50%, and 8 for 80%, whilst WTS is the best, respectively in 9, 13, and 14 cases. These strategies have equal performance in 5, 10, and 9 cases. In some cases there are no values, this is due to the difficult inherent to the problem, that is, none test case executed fails for the test budget under evaluation. On the other hand, considering the NTR values, we observe that VTS is the best strategy in 18 cases for time budget 10%, 17 for 50%, and 13 for 80%, whilst WTS is the best, respectively in 6, 13, and 15 cases. These strategies have equal performance in 6, 1, and 3 cases.

As observed for NAPFD, the greater the budgets the greater the WTS performance, and the opposite for VTS. However, the best RFCTC values highlighted do not provide good NTR values. This can be related to the standard deviation that is higher for RFCTC and WTS. For the variants fedora-undefined-sanitizer (fed-und) and tumbleweed-undefined-sanitizer (tumb-und), WTS has the best performance in most cases, across the time budgets. These variants are those with more builds. This suggests that increasing the number of commits WTS improves using the actual setting from our approach. In this sense, the sliding window size used could have impacted in the results. The use of a low size can improve the results for variants with few builds.

To visualize the learning behavior of the proposed approach, we analyze the accumulative NAPFD values. Figure 5 presents results from ubuntu-openssl_1.1.x-x86_64 (ubun-op) variant, as well as a comparison against the random prioritization. As we can observe, COLEMAN better fits the problem regardless of the strategy adopted, mainly when there are failures to occur (variation in the lines). Furthermore, we observe that WTS was a bit better than VTS

in the first commits, that is, WTS better mitigates the problem of beginning without learning.

To corroborate the quantitative analysis, we performed a qualitative analysis of the subject system variants and the potential benefits of using WTS as a prioritization strategy during the continuous integration of HCS. Table 5 describes the history of reused and failed test cases along with the different builds. In this table, we can observe the first build in which each variant was introduced (second column). The third column shows the number of test cases applied during the continuous integration in that first build. The number of reused and new test cases are presented in the fourth and fifth columns, respectively. The last column presents the number of the reused test cases that have failed when used for testing other variants previously to be used for that specific variant.

For the variants whose the test cases have never failed before the variant introduction, both VTS and WTS have the same behavior. On the other hand, for a new variant whose test cases have been failed before its introduction, failed history can be an important source of information for the prioritization of test cases, then WTS can bring benefits for the CI process, mainly considering time constraints. For instance, the variant mingw64 appears for the first time during build 45. All the 10 test cases of this variant were reused, since they were executed for testing other variants in previous build. One of these test cases failed 14 times in previous builds. In this case, this test case that revealed faults in previous build must be prioritized and executed before other test cases, since it is most likely to reveal faults.

Figure 6 presents the evolution of builds regarding the number of variants and test cases. The number of variants (blue line) represents the growth in the number of variants of libssh. Considering WTS, all the test cases considered in each build (red line) can be source of information during the learning process. Comparing to the average

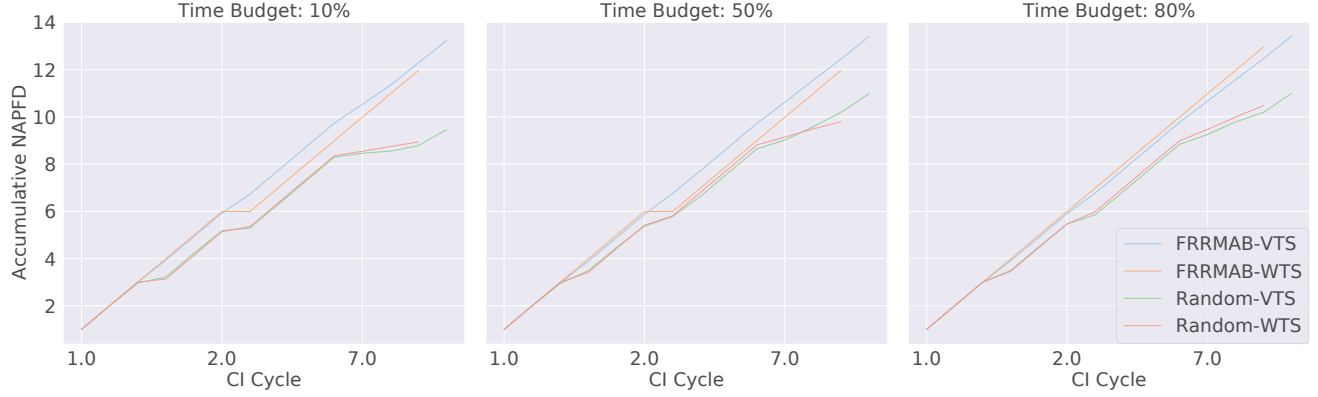


Figure 5: Accumulative NAPFD from ubuntu-openssl_1.1.x-x86_64 (ubun-op) variant.

Table 5: Reused Test Cases (TCs) and Failed Reused Test Cases along the build history.

Variant	First Build	Test Cases	Reused TCs	New TCs	Reused TCs Failed
CentOS7/openssl	2	17	0	17	0
Fedora/openssl	2	17	0	17	0
CentOS7/openssl 1.0.x/x86-64	3	17	17	0	2
Fedora/openssl 1.1.x/x86-64	4	17	17	0	3
Debian/openssl 1.0.x/aarch64	5	17	17	0	0
pages	18	29	17	12	0
address-sanitizer	22	29	29	0	18
undefined-sanitizer	22	29	29	0	47
fedora/openssl_1.1.x/x86-64	26	29	29	0	0
fedora/undefined-sanitizer	26	29	29	0	0
tumbleweed/openssl_1.1.x/x86-64	33	29	29	0	0
tumbleweed/undefined-sanitizer	33	29	29	0	0
fedora/mbedtls/x86-64	44	30	28	2	0
mingw64	45	10	10	0	14
mingw32	45	10	10	0	14
Debian.cross.mips-linux-gnu	64	19	18	1	0
fedora/openssl_1.1.x/x86-64/release	95	38	32	6	0
tumbleweed/openssl_1.1.x/x86-64/release	95	38	32	6	0
fedora/libcrypt/x86_64	131	40	38	2	0
centos7/openssl_1.0.x/x86_64	131	24	23	1	0
fedora/openssl_1.1.x/x86_64	131	40	38	2	0
fedora/mbedtls/x86_64	131	39	37	2	0
tumbleweed/openssl_1.1.x/x86_64/gcc	131	40	38	2	0
tumbleweed/openssl_1.1.x/x86_64/gcc7	131	40	38	2	0
tumbleweed/openssl_1.1.x/x86_64/clang	131	40	38	2	0
freebsd/x86_64	131	24	23	1	0
visualstudio/x86_64	188	16	11	5	0
visualstudio/x86	188	16	11	5	0
fedora/openssl_1.1.x/x86_64/minimal	197	40	39	1	0
fedora/openssl_1.1.x/x86_64/fips	258	55	46	9	0
ubuntu/openssl_1.1.x/x86_64	319	57	55	2	0

number of test cases per variant (yellow line), that represents the prioritization considering only the test cases of a specific variant, namely VTS, the source of information is very reduced.

RQ3: Based on the NAPDF results, we can not point the best strategy. WTS provides better results with less restrictive situation (more time budget), and VTS the opposite, as well as taking into account RFTC, NTR, and PR values. The use of historical data information from the test cases reused across the variants benefits WTS, which obtains better results than VTS in the first commits when there is no enough information to the learning.

6.4 Threats to Validity

Internal Validity: the parameter setting can be considered a threat. An ideal tuning of parameters was not performed in virtue of time constraints. To minimize this threat, we used the configuration based on previous work.

External Validity: we used only one subject system in this work. Thus, these results cannot be generalized. However, this system is a real-world system and the study provides some evidences towards an initial validation of our approach. New experiments should be performed to confirm these findings. Besides that, we believe that our study can be easily replicated, using the raw data analyzed and disseminated by the Open Science Framework (OSF).

Conclusion Validity: the randomness is a threat. The algorithms were executed 30 times but it is recommended they be executed 1000 times [2]. However, this is not possible due to the computational effort required. So, a larger number of executions should be considered. Another threat is related with the statistical tests used. To minimize this threat, we used tests commonly adopted for non-deterministic algorithms in software engineering problems [4]. Finally, the analysis was made with a set of quality indicators. These results may be different for other indicators.

7 RELATED WORK

In the literature we find mapping studies on regression testing and SPL engineering [11, 35]. Existing works explore the three basic regression testing techniques [43]: minimization [40], selection [16, 17, 41, 42], and prioritization [1, 12, 13] of test cases. These techniques are not excluding and can be combined [9]. These works do not focus CI particularities and constraints, but some of these approaches can be applied in a previous step to establish a test set for a build to be prioritized by our strategies.

Regarding TCP in CI environments we refer a recent mapping [30] that highlights some works for the HCS context [19, 20, 21, 22]. The approach of Marijan et al. [20, 21] uses historical test data to determine an optimal order of test cases to ensure feature coverage, early fault detection and execution time. An approach and a tool called TITAN is proposed in [22]. The tool implements test prioritization and minimization techniques, and provides test

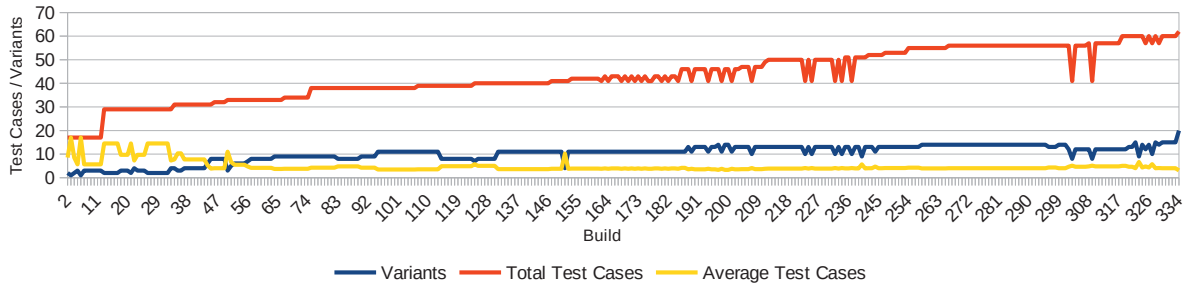


Figure 6: Number of variants and the number of test cases per build using Whole Test Set Strategy (WTS)

traceability and visualization. The idea is to obtain a high fault detection rate and low test execution. Another work [19] uses the coverage matrix of test cases and the fault detection history to identify redundant test cases that are not likely to detect faults. Their method minimizes a test suite by excluding redundant test cases. It is a learning algorithm that reduces test redundancy. The algorithm minimizes the test execution time by avoiding unnecessary test executions using coverage metrics and a fault-detection history. Again, additional information, such as feature coverage, is necessary. As firstly a minimization step is conducted, the prioritized set may not contain all available test cases.

To overcome such limitations, learning approaches based on historical failure data have been explored for TCP in CI. These approaches learn with past prioritizations, usually guided by a reward function [3] and deals properly with the volatility of the test cases. We can mention two approaches from the literature: (i) *RETECS* [37], a Reinforcement Learning-based approach that uses an agent, for instance, an Artificial Neural Network or a Tableau Representation, to interact with the CI environment and define an action (prioritization) to be applied according to a reward (feedback); and (ii) *COLEMAN* [28], which uses the MAB policy to establish the rewards. *COLEMAN* presented better performance than *RETECS* in experiments reported in the literature [28], and because of this was used in our study. But both strategies were not proposed specifically for the HCS context. Our work contributes to make possible the application of these approaches in CI of HCS, by introducing strategies that take into account particularities found in this context, such as the volatility of the variants.

8 CONCLUDING REMARKS

This work introduces two strategies for the application of a TCP learning-based approach, *COLEMAN*, in Continuous Integration of HCSs: the Variant Test Set Strategy (VTS) that relies on the test set specific for each variant, and the Whole Test Set Strategy (WTS) that prioritizes the test set composed by the union of the test cases of all variants. *COLEMAN* uses a MAB policy and a reward function to learn from the failure-history of test cases, addressing, in this way, the volatility problem, regarding test cases and variants that can be added or removed along the CI cycles.

We evaluated the strategies in a real-world HCS, using the FR-RMAB policy and TimeRank reward function, considering three time budgets, namely 10%, 50%, and 80%. The results show that the use of *COLEMAN* outperforms the use of a random prioritization in

terms of NAPFD, independently of the strategy and budget adopted. *COLEMAN* presents results that are statistically better or equal to the baseline in 93% of the cases. Furthermore, the strategies spend, in the worst case, just 0.04535 seconds to execute, what shows their applicability in the CI context. Both strategies present similar performance considering the indicators. But VTS performs better in the less restrictive scenario, i.e., time budget of 10%, and with WTS occurs the opposite. Furthermore, WTS better mitigates the problem of beginning without knowledge. Consequently, this strategy is adequate when there is a new variant to be tested.

Future work includes the application of the strategies (i) for other HCSs from different domains and with different number of variants; (ii) using *COLEMAN* with other policies, as well as other features such as test coverage, testers' preference, test case duration and, to consider in how many variants a test case fails; (iii) using other learning approaches such as the one based on Reinforcement Learning [37]; and (iv) using specific metrics to the variants.

ACKNOWLEDGMENTS

This research was partially funded by the Brazilian research agencies: CNPq (grants 408356/2018-9 and 305968/2018-1), Fundação Araucária – FAPPR (grant no. 51435), and CAPES.

REFERENCES

- [1] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-oriented product prioritization for similarity-based product-line testing. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 34–40.
- [2] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, USA, 1–10. <https://doi.org/10.1145/1985793.1985795>
- [3] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *42nd International Conference on Software Engineering (ICSE'20)*. ACM, New York, NY, USA. <https://doi.org/10.1145/3377811.3380369>
- [4] Thelma Elita Colanzi, Wesley Klewerton Guez Assunção, Paulo Roberto Farah, Silvia Regina Vergilio, and Giovanni Guizzo. 2019. A Review of Ten Years of the Symposium on Search-Based Software Engineering. In *Symposium on Search-Based Software Engineering*. Springer, Cham, 42–57.
- [5] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. 2008. An Empirical Study of the Effect of Time Constraints on the Cost-Benefits of Regression Testing. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 71–82.
- [6] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
- [7] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In

- 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, 235–245. <https://doi.org/10.1145/2635868.2635910>
- [8] Emelie Engström. 2010. Regression Test Selection and Product Line System Testing. In *3rd International Conference on Software Testing, Verification and Validation*. IEEE, 512–515.
- [9] Alireza Ensan, Ebrahim Bagheri, Mohse Asadi, Dragan Gasevic, and Yevgen Biletskiy. 2011. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *8th International Conference on Information Technology: New Generations*. IEEE, 291–298. <https://doi.org/10.1109/ITNG.2011.58>
- [10] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. 2016. Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing. In *8th International Symposium on Search Based Software Engineering*. Springer, Cham, 49–63.
- [11] Satendra Kumar and Rajkumar. 2016. Test case prioritization techniques for software product line: A survey. In *International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 884–889.
- [12] Remo Lachmann, Simon Boddig, Sascha Lity, Sandro Schulze, and Ina Schaefer. 2017. Risk-based integration testing of software product lines. In *11th International Workshop on Variability Modelling of Software-intensive Systems*. 52–59.
- [13] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Boddig, Sandro Schulze, and Ina Schaefer. 2015. Delta-oriented test case prioritization for integration testing of software product lines. In *19th International Conference on Software Product Line*. 81–90.
- [14] K. Li, A. Fialho, S. Kwong, and Q. Zhang. 2014. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on* 18, 1 (2014), 114–130.
- [15] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag, Berlin, Heidelberg.
- [16] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. 2019. Retest test selection for product-line regression testing of variants and versions of variants. *Journal of Systems and Software* 147 (2019), 46–63.
- [17] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. 2012. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *Tests and Proofs*. Springer, Berlin, Heidelberg, 67–82.
- [18] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* 18 (1947), 50–60.
- [19] Duscica Marijan, Arnaud Gotlieb, and Marius Liaaen. 2019. A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience* 49, 2 (2019), 192–213. <https://doi.org/10.1002/spe.2661>
- [20] Duscica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 540–543.
- [21] Duscica Marijan and Marius Liaaen. 2017. Test Prioritization with Optimally Balanced Configuration Coverage. In *IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 100–103.
- [22] Duscica Marijan, Marius Liaaen, Arnaud Gotlieb, Sagar Sen, and Carlo Ieva. 2017. TITAN: Test Suite Optimization for Highly Configurable Software. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 524–531. <https://doi.org/10.1109/ICST.2017.60>
- [23] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 643–54. <https://doi.org/10.1145/2884781.2884793>
- [24] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Balduino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469.
- [25] Mukelabai Mukelabai, Damir Nesjundefined, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York, USA, 155–166. <https://doi.org/10.1145/3238147.3238201>
- [26] Raiza Oliveira, Bruno Cafeo, and Andre Hora. 2019. On the Evolution of Feature Dependencies: An Exploratory Study of Preprocessor-Based Systems. In *13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, New York, NY, USA, 1–9. <https://doi.org/10.1145/3302333.3302342>
- [27] Jackson A. Prado Lima, William D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. 2020. Supplementary Material - Learning-based Prioritization of Test Cases in Continuous Integration of Highly-Configurable Software. <https://doi.org/10.17605/OSF.IO/SCD8M>
- [28] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments. *IEEE Transactions on Software Engineering* (2020), 12. <https://doi.org/10.1109/TS E.2020.2992428>
- [29] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. Multi-Armed Bandit Test Case Prioritization in Continuous Integration Environments: A Trade-off Analysis. In *5th Brazilian Symposium on Systematic and Automated Software Testing (SAST '20)*. ACM. <https://doi.org/10.1145/3425174.3425210>
- [30] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020), 106268. <https://doi.org/10.1016/j.infsof.2020.106268>
- [31] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. 2007. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *IEEE International Conference on Software Maintenance*. IEEE, 255–264. <https://doi.org/10.1109/ICSM.2007.4362638>
- [32] Herbert Robbins. 1985. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*. Springer, 169–177.
- [33] Gregg Rothermel. 2018. Improving Regression Testing in Continuous Integration Development Environments (Keynote). In *9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST 2018)*. ACM, New York, NY, USA, 1. <https://doi.org/10.1145/3278186.3281454>
- [34] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *IEEE International Conference on Software Maintenance (ICSM '99)*. IEEE Computer Society, 179.
- [35] Per Runeson and Emelie Engström. 2012. Chapter 7 - Regression Testing in Software Product Line Engineering. In *Advances in Computers*, Ali Hurson and Atif Memon (Eds.). Vol. 86. Elsevier, 223–263. <https://doi.org/10.1016/B978-0-12-396535-6.00007-7>
- [36] M. Shatin, M. Ali Babar, and L. Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943.
- [37] Helge Spieker, Arnaud Gotlieb, Duscica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 12–22. <https://doi.org/10.1145/3092703.3092709>
- [38] Andras Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (Jan. 2000), 101–132.
- [39] Alexander von Rhein, Alexander Grebhorn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *37th International Conference on Software Engineering - Volume 1 (ICSE 2015)*. IEEE, New York, USA, 178–188.
- [40] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. 2015. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software* 103 (2015), 370–391. <https://doi.org/10.1016/j.jss.2014.08.024>
- [41] Shuai Wang, Shaukat Ali, Arnaud Gotlieb, and Marius Liaaen. 2016. A systematic test case selection methodology for product lines: results and insights from an industrial case study. *Empirical Software Engineering* 21 (2016), 1586–1622.
- [42] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. 2013. Continuous Test Suite Augmentation in Software Product Lines. In *17th International Software Product Line Conference (SPLC '13)*. Association for Computing Machinery, New York, NY, USA, 52–61. <https://doi.org/10.1145/2491627.2491650>
- [43] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software: Testing, Verification, and Reliability* 22, 2 (March 2012), 67–120. <https://doi.org/10.1002/stvr.430>

APPENDIX F – COST-EFFECTIVE LEARNING-BASED STRATEGIES FOR TCPCI OF HCS

Cost-effective learning-based strategies for test case prioritization in Continuous Integration of Highly-Configurable Software

Jackson A. Prado Lima
DInf, Federal University of
Paraná
Curitiba, Brazil
jackson.lima@ufpr.br

Willian D. F.
Mendonça
DInf, Federal University of
Paraná
Curitiba, Brazil
willian.mendonca@ufpr.br

Silvia R. Vergilio
DInf, Federal University of
Paraná
Curitiba, Brazil
silvia@inf.ufpr.br

Wesley K. G. Assunção
Pontifical Catholic
University of Rio de Janeiro
Rio de Janeiro, Brazil
wassuncao@inf.puc-rio.br

ABSTRACT

Highly-Configurable Software (HCSs) testing is usually costly, as a significant number of variants need to be tested. This becomes more problematic when Continuous Integration (CI) practices are adopted. CI leads the software to be integrated and tested multiple times a day, subject to time constraints (budgets). To address CI challenges, a learning-based test case prioritization approach named COLEMAN has been successfully applied. COLEMAN deals with test case volatility, in which some test cases can be included/removed over the CI cycles. Nevertheless, such an approach does not consider HCS particularities such as, by analogy, the volatility of variants. Given such a context, this work introduces two strategies for applying COLEMAN in the CI of HCS: the Variant Test Set Strategy (VTS) that relies on the test set specific for each variant; and the Whole Test Set Strategy (WTS) that prioritizes the test set composed by the union of the test cases of all variants. Both strategies are applied to two real-world HCSs, considering three test budgets. Independently of the time budget, the proposed strategies using COLEMAN have the best performance in comparison with solutions generated randomly and by another learning approach from the literature. Moreover, COLEMAN produces, in more than 92% of the cases, reasonable solutions that are near to the optimal solutions obtained by a deterministic approach. Both strategies spend less than one second to execute. WTS provides better results in the less restrictive budgets, and VTS the opposite. WTS seems to better mitigate the problem of beginning without knowledge, and is more suitable when a new variant to be tested is added.

KEYWORDS

Test Case Prioritization, Software Product Line, Continuous Integration, Highly-Configurable Software

1 INTRODUCTION

To succeed, software systems must consider and operate over different user's needs, mostly related to the different business domains, organizational processes, environmental restrictions, and specialized hardware devices [4]. In this scenario, Highly Configurable Software (HCS) provides adaptable and flexible solutions to complex and real-world problems. HCS is usually implemented by using different configuration options – applying strategies such as conditional compilation, conditional execution, or build systems – to create custom system products, a.k.a. variants [32, 50].

To ensure HCS quality is a fundamental issue, and some specific approaches are needed. For instance, HCS testing can be more complex and costly, as many variants usually need to be tested, and many test cases overlap. This becomes more problematic if Continuous Integration (CI) practices are adopted [25]. CI environments have become popular in the industry to allow automatic integration, build, and testing of software projects, created by different developers/teams collaboratively [47]. CI leads the software to be integrated and tested multiple times a day to detect integration errors as quickly as possible [9]. However, running a large set of test cases can take many minutes or even hours [44]. This implies costs, requires automated tools, and makes the continuous regression testing of HCS more challenging [11, 52].

As we can see in CI, it is fundamental to perform regression testing in a very cost-effective way, providing rapid test feedback on software failures [53]. Another essential point to consider is that, in a company, multiple projects may share the same CI environment, imposing time testing constraints [8]. The test must run a restricted slot of time, referred to as *test budget*. Therefore, some traditional regression testing approaches are not suitable for CI, mainly those for selection and minimization of test cases that rely on code coverage, code instrumentation, or search-based techniques that take a long time to execute [53]. In this sense, Test Case Prioritization (TCP) is more popular and used [8]. TCP techniques improve the cost-effectiveness of regression testing by ordering test cases to allow early execution of the most important ones, generally those test cases with a high probability of revealing faults [10]. Considering the constraints of test budgets, early fault detection is essential because when a test case fails, test execution can be ended, and fewer resources are spent.

Approaches have appeared to adapt TCP techniques for the CI context (TCPCI) [39]. Nevertheless, the great majority are not adaptive, that is, they do not learn with past prioritizations; they do not consider test case volatility, characteristically associated with the fact that test cases may be added or removed (discontinued) over the CI cycles. Learning approaches based on historical failure data have been proposed to overcome some of these limitations. Ranking-to-learn approaches learn from past prioritization, based on the rewards obtained from the feedback of previously used ranks. The main idea is to maximize the rewards [37, 48]. These approaches are more robust regarding the volatility of the test cases, code changes, and the number of failing tests [3]. However, the challenge is to search for early fault detection in the failure-history of past test

cases, but also to explore new test cases. This is related to the Exploration versus Exploitation (EvE) dilemma, and is a consequence of the test budget, since whether only error-prone test cases are considered without diversity, some test cases can never be executed. COLEMAN (*Combinatorial VOLatile Multi-Armed BANdit*) [37] is a promising approach to deal with the EvE dilemma. COLEMAN formulates TCPCI as a Multi-Armed Bandit (MAB) problem [43]. A test case is an arm and at each time/build multiple arms are selected. In addition, the arms available at each time may change dynamically over time. In this way, it learns with the feedback from the application of the test cases, incorporating diversity in the test suite prioritization. COLEMAN has presented better results in comparison with other learning approaches and can be considered the state-of-the-art ranking-to-learn approach [37]. However, COLEMAN and other existing learning approaches do not consider the HCS particularities.

The limitation of existing TCPCI approaches applied in the particular context of HCS, is that they do not adequately address test case volatility or are model-dependent [15, 25, 27, 29]. To deal with the EvE problem, most of them use, besides the failure-history, other measures that rely on code instrumentation or require additional information, such as code or feature coverage. This can be time-consuming, and either maintaining this information and updated models can be challenging.

Moreover, in the HCS context, we have another particularity, that is, by analogy, the volatility of variants that have different configuration options. Each variant can be seen as a system to be individually tested, however, having test cases that are common to or reused from other variants. Also, some variants can be included or discontinued over the CI cycles. To mitigate this problem, in a previous work [35], we proposed two strategies for the application of learning-based approaches, such as COLEMAN, in the CI of HCS: (i) the Variant Test Set Strategy (VTS) that relies on the test set specific for each variant; and (ii) the Whole Test Set Strategy (WTS) that prioritizes the test set composed by the union of the test cases of all variants.

The proposed strategies were applied to the system LIBSSH and evaluated regarding some indicators of early fault detection and time reduction. The obtained results showed that the use of such strategies with COLEMAN outperforms a random strategy. These preliminary results motivated the present work, which extends previous work by adding new evaluations and comparisons with the other two approaches. Adopting the same procedures, we added to the analysis a new HCS, namely DUNE, as well as new quality indicators to evaluate the strategies. Moreover, we performed a comparison with another approach from the literature, namely RETECS [48], based on reinforcement learning. Moreover, all the approaches - COLEMAN, RETECS and random - are also evaluated against a deterministic approach. Such an approach determines the optimal prioritization based on a priori knowledge of the tests results. The main idea is to determine how far the solutions produced by the strategies using each one of the approaches are from the optimal solutions produced by the deterministic approach.

The obtained results show that the proposed strategies using COLEMAN are very cost-effective. COLEMAN produce, in the great majority of the cases and independently of the time budget, the best solutions in comparison with the random and RETECS approaches.

The solutions produced by COLEMAN are considered reasonable, that is, near to the optimal solutions in 92% of the cases. Moreover, both strategies spend less than one second to execute, i.e., they are applicable considering the CI Cycles. In this way, the main contributions of this work are:

- A deeper evaluation on the use of the proposed strategies by adding a new system to our evaluation. This contributed to corroborating some previous results, which are also valid for the system added, and to derive some guidelines for using the strategies. WTS provides better results in a less restrictive situation where more time and resources are available, and VTS the opposite. The use of historical data information from the test cases reused across the variants benefits WTS. Because of this, WTS obtains better results than VTS in the first commits when there is no enough information to the learning and when a new variant is added.
- Evaluation on the use of another learning approach in comparison with COLEMAN. The obtained results show COLEMAN is the best option in comparison with RETECS, and corroborate the MAB-based approach to allow mitigating the variant and test cases volatility problem, learning from the failure history of reused test cases along with the CI cycles, and combining exploration and exploitation.
- Evaluation results that show the strategies used with COLEMAN are very cost-effective. The solutions generated by COLEMAN are very close to the optimal solutions produced by the deterministic approach. Moreover, they are applicable in practice, considering the CI cycles, spending only a few seconds to run.
- A set of findings whose implications are analyzed in terms of practice and research aspects. They can serve to direct future research in this subject.
- A public repository with the data used in this work and the implementation for mining the CI information, which allows replication and future research [36].

This paper is structured as follows. Section 2 presents a motivating example. Section 3 reviews TCPCI approaches and Subsection 3.1 describes COLEMAN, the learning-based approach adopted in this work. Section 4 introduces the strategies proposed for the HCS context. Section 5 describes how the evaluation was conducted. Results are presented and analyzed in Section 6. Section 6.6 discusses some findings and implications. Section 6.7 presents some threats to the validity of our results. Section 7 overviews related work, TCP approaches for SPLs and HCSs, in general and specific for the CI context. Section 8 presents concluding remarks.

2 MOTIVATING EXAMPLE

To highlight the importance of having TCP during CI of HCSs, we provide a real-world example from the LIBSSH system, which is one of the subject systems used in our study (see Section 5.1). In the commit #17b518a6¹ from the LIBSSH system, the developers included a new feature that adds support for new OpenSSH keytypes, as presented in Figure 1. The implementation of this new feature implied changes on seven files with 245 additions and 13 deletions.

¹<https://gitlab.com/libssh/libssh-mirror/-/commit/17b518a677c92d943cf016b81272ec10ee1ca368>

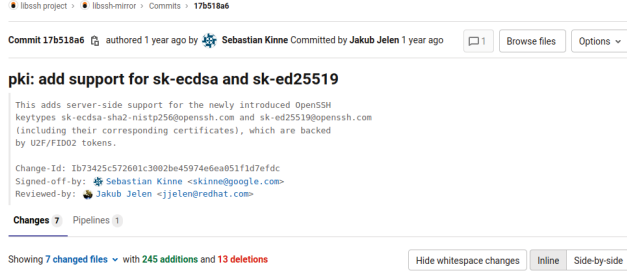


Figure 1: New LIBSSH feature introduced on Feb 11, 2020 to support sk-ecdsa and sk-ed25519 OpenSSH keytypes.

In order to test the newly introduced feature, Pipeline #116809311² was triggered. Figure 2 presents a piece of information from this pipeline that was composed of 27 jobs for the master branch, in which each job is in charge of testing one variant. In total, the CI with the 27 variants of LIBSSH took 23 minutes and 39 seconds. In the figure, we can notice that two jobs failed during the CI, namely for variants ubuntu/openssl_1.1.x/x86_64 and visualstudio/x86_64. This implies that some test cases failed.

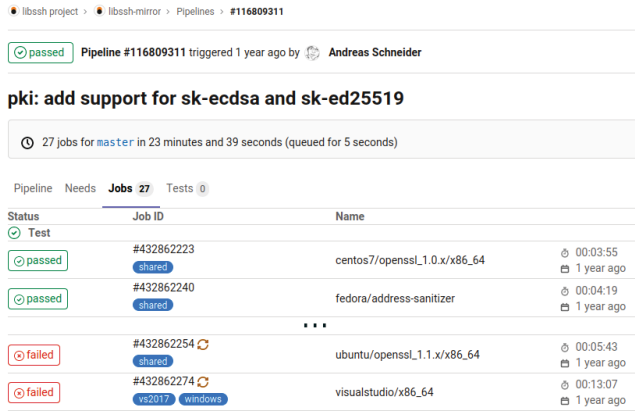


Figure 2: CI pipeline for feature pki: add support for sk-ecdsa and sk-ed25519 with 27 jobs/variants for the master branch.

For a more in-depth analysis, we focus on the fails of variants ubuntu/openssl_1.1.x/x86_64 in the job #432862254³. Figure 3 presents a snippet of the build log from the test of this variant. In the figure, we can notice that the test case #39 (on line 837) has failed. Based on the testing order defined for this job by the developers/testers of LIBSSH, the test activity took 122.63 seconds until reaching the test case that failed. In total, this variant has 57 test cases, and it takes 197.01 seconds to run all tests. Considering this scenario, this job's testing activity consumed 62% of the runtime to reach the failed test case. Supposing we have a budget constraint for testing LIBSSH and we defined 50% of the total runtime and no

²<https://gitlab.com/libssh/libssh-mirror/pipelines/116809311/builds>

³<https://gitlab.com/libssh/libssh-mirror/-/jobs/432862254>

prioritization, the test case that reveals a fail would not be executed, without finding the bug related to the inclusion of the new feature.

This illustrative example shows the CI environment for a new feature of the LIBSSH system. We can see how time consuming the CI process is when there are many HCS variants. Also, we described the problem of not having a proper prioritized testing order in cases of budget constraints. This example highlights and motivates the importance of having a prioritization strategy to execute earlier those test cases that are more likely to reveal faults.

3 TEST CASE PRIORITIZATION IN CI

The last section presented the importance of test case orders. Nevertheless, as we mentioned before, TCP CI involves challenging particularities such as time constraints (test budgets) and the volatility of test cases. Due to these particularities, some traditional approaches, such as the ones based on search and coverage, are not suitable for CI, because they take time to execute [10].

More recently, other approaches have been proposed to the CI scenario, as reported by a study mapping on TCP CI [39]. However, most of them [5, 14, 24, 27, 28] require code analysis, which can be costly. Moreover, they do not address the main characteristics of CI environments. For instance, they do not consider test case volatility, a characteristic associated with the fact that test cases may be added and/or removed over the cycles. They are not adaptive, that is, they do not learn with past prioritizations.

To overcome such limitations, learning approaches based on historical failure data have been proposed. Bertolino et al. [3] distinguish two kinds of TCP learning-based approaches. The first one, named Learning-to-Rank, uses supervised learning to train a model based on some test features. The model is then used to rank test sets in future commits. The problem with these strategies is that the model may no longer be representative when the commit context changes. The second kind, named Ranking-to-Learn, is more suitable to the dynamic CI context. This strategy learns based on the rewards obtained from the feedback of previously used ranks. The main idea is to maximize the rewards. Ranking-to-learn approaches present some advantages. They are more robust regarding the volatility of the test cases, code changes, and the number of failing tests. Because of this, the focus of our work is on this kind of approach.

In the literature, we can find two ranking-to-learn approaches in the CI context: RETECS[48] (*Reinforced Test Case Selection*), an approach based on Reinforcement Learning; and (ii) COLEMAN[37] (*Combinatorial VOLatile Multi-Armed BANDit*), an approach based on Multi-Armed Bandit. In experiments reported in the literature [37], COLEMAN presented better performance than RETECS. Because of this, the strategies proposed adopt COLEMAN as learning approach, and RETECS is used as a baseline for comparison.

3.1 Adopted Approach

COLEMAN, the approach adopted in our study, formulates the TCP CI problem as a Multi-Armed Bandit (MAB) problem. MAB problems are sequential decision problems related to the scenario in which a player plays on a set of slot machines (or arms/actions) that even being identical produce different gains [43]. After a player pulls one of the arms in a turn, a reward is received from some unknown

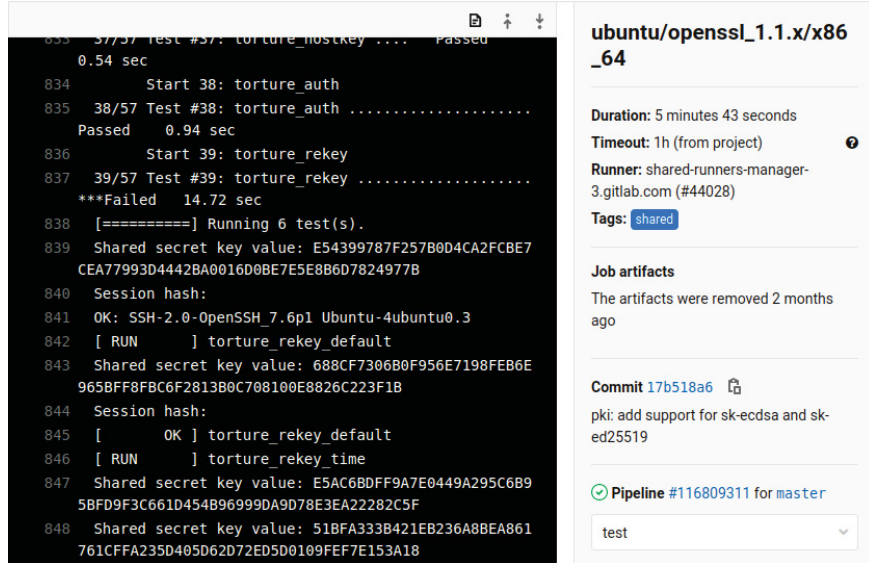


Figure 3: Test #39 failed for ubuntu/openssl_1.1.x/x86_64 after 122.63 seconds of testing.

distribution, aiming to maximize the sum of the rewards. To this end, a MAB policy is applied to choose, at each time, the next arm to pull based on previously observed rewards and decisions.

The formulation of COLEMAN uses a test case as an arm to be pulled, and it encompasses two MAB variations, namely combinatorial and volatile, to deal with the dynamic nature of our problem. In the combinatorial variation, at each turn (commit/build/CI Cycle), a MAB policy selects all arms (test cases) available instead of one. According to the order in which the policy selects the arms, the prioritization is defined. The volatile variation deal with the test case volatility. In this case, only the test set available at each commit is used by the policy.

Only historical failure data is required. No further detail about the system under test is needed, such as code coverage or code instrumentation. Figure 4 shows how COLEMAN interacts with the CI environment. In such an environment, teams work continuously integrating code and making smaller code commits every day, usually monitored by a CI server. When a change occurs, the CI server clones this code, builds it, and runs the testing processes. When the entire process is finished, a report is generated by the CI server, and the developers are informed. COLEMAN acts after a successful build, in the test phase the approach prioritizes the test case set available T_c to be used during the test case execution.

For each build triggered by a commit c , a test case set T_c is available, as well as a test budget. When the budget is smaller than the total time required to execute T_c , our approach is then used to obtain a prioritized test set T'_c . The idea is that the most relevant test cases, i.e., the ones that fail, are executed first. A MAB policy is used to determine the most relevant test cases. After the test execution, the policy receives a reward (feedback) provided by a reward function. The reward value is used in the credit assignment procedure to set individual rewards for each test. Based on rewards, the policy adapts its experience for future actions (*online learning*).

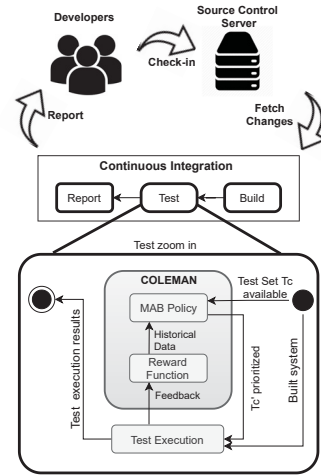


Figure 4: Overview of the COLEMAN interaction with the CI environment.

COLEMAN can be used with different MAB policies and reward functions. But as presented and evaluated in previous studies [37], COLEMAN obtained the best performance using TimeRank (*Time-Ranked*) function and FRRMAB (*Fitness-Rate-Rank based on Multi-Armed Bandit*) policy [20]. Thus, for each test case, FRRMAB policy considers the history of rewards whilst other policies use cumulative rewards. For this reason, we used FRRMAB and TimeRank in this study. They are described next.

3.1.1 FRRMAB policy. The FRRMAB policy evaluates all test cases from a test set available. It selects the best one considering an empirically estimated value based on a range that depends on the number of times that has been applied previously. For this, two

procedures are used: credit assignment and operator (arm/action) selection.

Credit Assignment refers to a reward procedure that takes into account the impact observed in the most recent applications. During the credit assignment, FRRMAB adopts a rank-based method that uses the Fitness Improvement Rate (FIR) method. In our context, the FIR value is obtained by the reward function. The FIR value is stored in a given Sliding Window (W) organized as a first-in, first-out queue that is used to evaluate the W recent applications. In this way, it is considered the last W commits as historical test data. Such behavior allows to exploit the best test cases. If a new test case appears, a zero is set for the reward value, once that in this case there is no *test case history*. On the other hand, if a test case is removed in the current cycle (commit), we remove its history. The FRRMAB final value is corrected by a decaying factor.

Operator Selection takes into account the number of times that a test case appeared in the last W commits. This value is used to explore new test cases or test cases with few executions.

In order to consider the combinatorial MAB behavior, COLEMAN adapts the policy to evaluate all the test cases (arms), and order them, putting the best test case in the top, followed by the second best one, and so on. When more than one test case has the same *performance*, the order among them is defined randomly. On the other hand, the MAB policy was also adapted to consider only the test cases available in time t and ignore the other ones from the previous time. This modification allows considering the dynamic environment (volatility) of the test cases.

3.1.2 Reward Function. The *TimeRank* function, adopted in our study, is defined in Equation 1. This function is based on the rank of t'_c in $T'_c \forall t'_c \in T'_c$, where T'^{fail} is composed by the failing test cases from T'_c ; *RNFail* returns 1 if t'_c failed and 0 otherwise. The *prec*(t'_{c_1}, t'_{c_2}) function returns 1 if the position of t'_{c_1} is lower than the position of t'_{c_2} . The idea is to evaluate whether failing test cases are ranked in the first positions in T'_c . To this end, a test case t'_c that does not fail and precedes failing ones are penalized by their early scheduling. A non-failed test case receives a reward given by the accumulated number of test cases which failed until its position in the prioritization rank, that is, it receives a reward decreased by the number of failing test cases ranked after it.

$$TimeRank(t'_c) = |T'^{fail}| - [-(RNFail(t'_c)) \times \sum_{i=1}^{|T'^{fail}|} prec(t'_c, t'_{c_i})] \quad (1)$$

4 PROPOSED STRATEGIES

As we mentioned, the HCS context has some particularities, namely variant volatility. To deal with variant volatility in the CI of HCS, in this section, we introduce two strategies. They allow the application of COLEMAN for HCS and learn from past failure history.

For the integration of a commit c of an HCS S , COLEMAN was adapted to consider that there is a set V of n variants. For each variant $v_i \in V$ a set of test cases $T_{v_i,c}$ is available. That is, in this study we consider that there is only one test set by variant. COLEMAN receives as input n test sets, $\{T_{v_1,c}, T_{v_2,c}, \dots, T_{v_n,c}\}$, and produces n prioritized test sets, $\{T'_{v_1,c}, T'_{v_2,c}, \dots, T'_{v_n,c}\}$. The value n can vary depending on

the number of existing variants of S when c is committed. For the COLEMAN prioritization, we propose two strategies to deal with a set of variants and volatility, as illustrated in Figure 5.

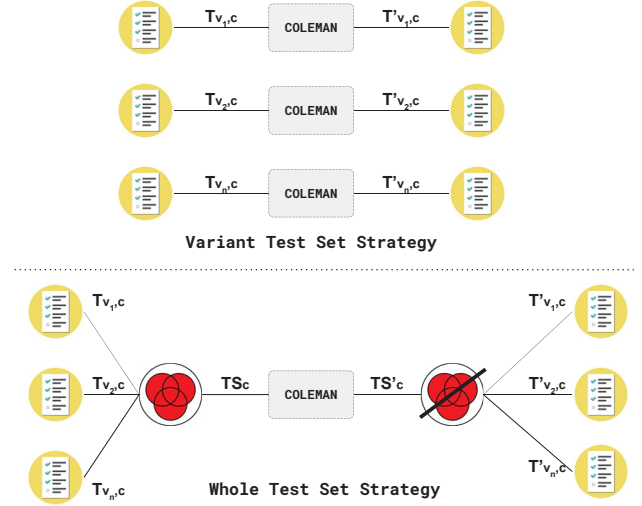


Figure 5: Strategies to deal with HCS variants and their volatility using COLEMAN.

1. Variant Test Set Strategy (VTS): in this strategy, presented at the top of Figure 5, COLEMAN is applied n times for each c treating each variant independently. The i^{th} application has as input the set T_{v_i,c_i} and as output the prioritized set T'_{v_i,c_i} . When a new variant of S is introduced, no history information is available.

2. Whole Test Set Strategy (WTS): for this strategy, presented at the bottom of Figure 5, COLEMAN is applied only once for each c and has as input only one test set $TS_c = \bigcup_{i=1}^n T_{v_i,c}$ composed by the union of all test sets of all variants under test. A test case t in TS_c can be common/reused to more than one variant, but it appears in TS only once. The status of t is set to failed whether t failed in at least one variant in which the test case have been executed previously. To calculate the total time required to execute TS we set the duration of t with the maximum execution time for t considering all variants. The output is a prioritized set TS'_c . The sets $T'_{v_i,c}$ are then generated, by selecting from TS'_c only the same test cases that belong to $T_{v_i,c}$ but keeping the prioritization order.

The main advantage of WTS is that if a new variant appears, it can be tested based on the historical information collected from the other ones. In this way, mitigating the problem of beginning without knowledge (learning) and adapting to changes in the execution environment, either by test case volatility or variant volatility.

5 EVALUATION SETUP

This section describes the evaluation setup of our study. The main goal is to evaluate the proposed strategies regarding early fault detection and time reduction. Based on this goal, we derived five research questions (RQs). The first three RQs evaluate our proposed strategies using COLEMAN, in comparison with three other

approaches: random, RETECS, and deterministic. The other two RQs evaluate the applicability of the strategies in the CI cycles and compare their performance⁴.

RQ1: *What is the performance of the strategies VTS and WTS when a random approach is used in comparison with COLEMAN?* This question compares the results obtained by the strategies VTS and WTS using COLEMAN with a prioritization order generated randomly. Random prioritization is one of the most applicable and fundamental TCP technique [57]. The comparison is performed using an indicator of early fault detection.

RQ2: *What is the performance of the strategies VTS and WTS using COLEMAN in comparison with RETECS?* This question compares the results obtained by the strategies VTS and WTS using COLEMAN with another learning approach from literature RETECS. RETECS was chosen because it is specific for CI. Other existing approaches are model-dependent or required code instrumentation or annotations that make difficult their application. The comparison also takes into account early fault detection.

RQ3: *How far are the solutions of all adopted approaches from optimal solutions obtained using a deterministic approach?* This question allows us to know how reasonable the solutions obtained by the three approaches - COLEMAN, RETECS, and random - are. To answer this question, we use a prioritization generated deterministically. The deterministic approach uses a priori knowledge about the tests results. This knowledge is not available in the moment the prioritization is performed, because it is generated only after the test cases execution. Then, such an approach can not be applied in practice, but gives us a measure to evaluate the quality of the solutions.

RQ4: *Are the strategies VTS and WTS applicable for HCSs in the CI development context?* This question investigates whether the time spent in the prioritization is acceptable considering CI Cycles (commits) and if the strategies proposed can be useful in practice.

RQ5: *What is the best strategy for test case prioritization in the context of HCS?* This question aims to compare the proposed strategies, VTS and WTS. The idea is to evaluate their ability regarding early fault detection, and percentage of reduced time, to provide directions for their adoption.

5.1 Subject Systems

In this section, we describe the systems under test (SUT) used in this study, namely LIBSSH and DUNE. These systems have been already used in the literature on the topic of HCS and SPLs [13, 30, 31, 33, 41, 42]. They have a representative number of variants and historical to a preliminary evaluation of our strategies. These systems are hosted at GitLab⁵, which provides an environment with control version system and CI pipelines.

⁴The systems and data collected, as well as the values found for each metric with statistical test results and plots, are available in our supplementary material [36].

⁵LIBSSH at <https://gitlab.com/libssh/libssh-mirror> and DUNE at <https://gitlab.dune-project.org>

The logs of the CI pipeline jobs are the source of information for COLEMAN and RETECS. Figure 6 presents a real example of log from a job⁶ of LIBSSH. The log describes information related to the configuration, build, and execution of test cases for a variant. This latter is the basis to collect input information for COLEMAN. We can observe in the figure that the execution of test cases started on line 759, and the result of the first test case is seen on line 761. This line 761 has the three pieces of information that are collected: (i) The name of the test case, in this example “torture_buffer”; (ii) the status of the execution, which in this case is “Passed”, but can also be “Failed”, as we can see in the bottom of the figure, on line 837; and (iii) execution time that in this test case on 761 is equal to “0.26 sec”.

```

758 [100%] Built target test_server
759 Test project /builds/libssh/libssh-mirror/obj
760 Start 1: torture_buffer
761 1/57 Test #1: torture_buffer ..... Passed 0.26 sec
762 Start 2: torture_bytearray
763 2/57 Test #2: torture_bytearray ..... Passed 0.00 sec
764 Start 3: torture_callbacks
765 3/57 Test #3: torture_callbacks ..... Passed 0.00 sec
766 Start 4: torture_crypto
767 4/57 Test #4: torture_crypto ..... Passed 0.00 sec
768 Start 5: torture_init
...
837 39/57 Test #39: torture_rekey .....***Failed 14.72 sec
838 [=====] Running 6 test(s).
839 Shared secret key value: E54399787F257B0D4CA2FCBE7CEA77993D4442BA001600
840 BE7E5E8B6D7824977B
841 Session hash:
842 OK: SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
843 [ RUN ] torture_rekey_default
844 Shared secret key value: 688CF7306B0F956E7198FEB6E965BFF8FBC6F2813B0C70
845 8100E8826C223F1B

```

Figure 6: Example of Job Log

To collect the CI build history, we developed the tool name GitLabCI Torrent, which the source code is available at GitHub⁷. We choose these systems because they provide detailed output traces (more verbose), allowing GitLabCI Torrent to extract the historical test data information adequately. Both systems were mined on January, 2021.

The SSH library (LIBSSH⁸) is an open-source C multiplatform library implementing the SSHv2 protocol on client and server side. This library is statically configurable with the C preprocessor, and it is designed to remotely execute programs, transfer files, use a secure and transparent tunnel, manage public keys, and the like. In this system, we identified 43 variants on GitLab. From these variants, we selected 34 that have at least one failed test case in their history of builds.

The Distributed and Unified Numerics Environment (DUNE⁹) system is a modular tool for solving partial differential equations with grid-based methods. It supports the easy implementation of methods like Finite Elements, Finite Volumes, and also Finite Differences. Using C++ techniques, DUNE allows one to use very implementations of the same concept, i.e., grid and solver, under a common interface with a very low overhead. Similarly to the LIBSSH system, we identified 33 variants for this system, and we selected 29.

⁶<https://gitlab.com/libssh/libssh-mirror/-/jobs/432862254#L837>

⁷<https://github.com/jacksonpradolima/gitlabci-torrent>

⁸<https://www.libssh.org/>

⁹<https://www.dune-project.org/>

Information about the LIBSSH and DUNE systems are presented in Tables 1 and 2, respectively. The first line of the tables presents information of the set composed by the union of the test cases related to all variants, which is considered to evaluate WTS. The first column shows the variant name. The second column shows the period that each variant was used. The third column presents the total of builds identified. Only valid builds (success or fail) were considered, that is, we discarded builds with some problem, for instance, canceled builds. The fourth column shows the total of failures found, and in parentheses the number of builds in which at least one test failed. The fifth column presents a graph concerning the faults by cycle. The sixth column shows the number of different (unique) test cases identified from build logs, and in parentheses the range of test cases executed in the builds. The seventh column presents a graph concerning the test case volatility. The last columns present the mean (\pm standard deviation) duration in minutes of the CI Cycles and the interval between them.

5.2 Quality Indicators

We adopted indicators from TCP literature [37, 38, 40] regarding early detection, time reduction, and accuracy. They are defined as follows.

Normalized Average Percentage of Fault Detected (NAPFD) [40] is an extension of the Average Percentage of Faults Detected (APFD) [45]. APFD measures how fast a set of prioritized test cases (T') can success on detecting faults in the program under tested. APFD values range from zero to one and is computed from the weighted average of the percentage of detected faults. Higher values indicate that the faults are detected faster using fewer test cases. On the other hand, NAPFD metric is adequate for prioritization of test cases when not all of them are executed, and some faults can be undetected. NAPFD, in addition to APFD, considers the ratio between detected and detectable faults within T . Equation 2 describes how to compute NAPFD, where m is the number of faults detected by all test cases; $rank(T'_i)$ is the position of T'_i in T' . If T'_i did not reveal faults, then it is set to $T'_i = 0$. n is the number of tests cases in T' and p is the number of faults detected by T' divided by m .

$$NAPFD(T') = p - \frac{\sum_1^n rank(T'_i)}{m \times n} \quad \frac{p}{2n} \quad (2)$$

Normalized Time Reduction (NTR) [37] evaluates the capability to reduce the time spent in a CI Cycle. It measures the difference between the time spent to reach the first failed test case r_t and the time to execute all tests \hat{r}_t . In such a metric, only commits that fail CI^{fail} are considered. The values range from 0 to 1, in which higher values represent a higher test time reduction. Its calculation is given in Equation 3.

$$NTR(\mathcal{A}) = \frac{\sum_{t=1}^{CI^{fail}} (\hat{r}_t - r_t)}{\sum_{t=1}^{CI^{fail}} (\hat{r}_t)} \quad (3)$$

Prioritization Time (PT) takes into account the runtime spent by an approach to perform the prioritization in each commit. This value is used to provide an indicative of the applicability of the proposed

strategies in real scenarios. In our experiments, we measure PT in seconds.

Root Mean Square Error (RMSE) [38] is used to observe the difference between the predicted and the observed values, for instance, concerning NAPFD values. In this study, we consider the differences found by the approaches (\hat{s}_t) and the optimal values T' (s_t) found by a deterministic approach. For an algorithm \mathcal{A} , we compute the difference for each CI Cycle (commit) t in relation to the amount of CI Cycles CI in a system. The most accurate algorithm is the one with smallest RMSE. Equation 4 shows how to calculate RMSE.

$$RMSE(\mathcal{A}) = \sqrt{\frac{\sum_{t=1}^{CI} (\hat{s}_t - s_t)^2}{CI}} \quad (4)$$

In order to represent how far the solutions found by an algorithm are from the optimal solutions, an ordinal scale of RMSE magnitude is used, as follows:

$$RMSE \text{ Magnitude} = \begin{cases} \text{very near} & \text{if } RMSE < 0.15 \\ \text{near} & \text{if } 0.15 \leq RMSE < 0.23 \\ \text{reasonable} & \text{if } 0.23 \leq RMSE < 0.30 \\ \text{far} & \text{if } 0.30 \leq RMSE < 0.35 \\ \text{very far} & \text{if } 0.35 \leq RMSE \end{cases} \quad (5)$$

where (i) the *very near* category represents an approximated optimal performance; (ii) the *near* category represents reaching optimal performance, and that some improvements are required; (iii) the *reasonable* category represents the minimum acceptable performance. Solutions in this category are acceptable and are related to the cases in which the system behavior, or possibly the constraints, can make the prioritization task hard; (iv) the *far* category represents unsatisfactory performance, and that meaningful improvements are required; and (v) the *very far* category includes solutions that are far away from to be useful and considered reasonable. In our analysis, the solutions are considered acceptable if they are at least reasonable, that is, if $RMSE < 0.3$.

5.3 Applying Approaches

We defined three configurations of test budgets, namely 10%, 50%, and 80% of the execution time of the overall test set available in each commit. These different test budgets allow us to investigate the behavior of each approach and strategy during the TCP in CI process. Below we describe how the three approaches evaluated in this study were executed.

COLEMAN: To apply COLEMAN we adopted the same settings reported in previous work [37]. We used the same configuration for FRRMAB: sliding window size W equals to 100, the coefficient C to balance exploration and exploitation equals to 0.3, and decayed factor equals to 1.

RETECS: This approach was executed with an Artificial Neural Network (ANN) [7]. We adopted default values defined by the authors and the version that is available online [48]. We set the parameters for Hidden Nodes, Replay Memory, and Replay Batch Size, with, respectively, 12, 10000, and 1000. ANN was chosen because it presented the best performance in the referred work.

Table 1: System Information - LIBSSH

Name	Period	Builds	Faults	Faults/Cycle	Tests	Volatility	Duration	Interval
Total	2018/04/12-2021/01/20	401	281 (159)		64 (1 - 60)		0.3206 (1.632)	1144.2097 (472.297)
CentOS7-openssl	2018/04/12-2018/04/12	1	1 (1)		17 (17 - 17)		0.0237 (0.042)	-
CentOS7-openssl 1.0.x-x86-64	2018/04/12-2018/04/18	22	8 (7)		29 (17 - 29)		0.0104 (0.02)	1320.7984 (287.806)
Debian-openssl 1.0.x-aarch64	2018/04/12-2018/04/13	8	1 (1)		17 (17 - 17)		0.0431 (0.083)	1298.6738 (306.178)
Debian_cross_mips-linux-gnu	2018/07/02-2018/12/24	131	7 (7)		29 (19 - 29)		1.3815 (4.122)	956.783 (499.357)
Fedora-libcrypt-x86-64	2018/04/12-2018/04/13	8	4 (2)		29 (17 - 29)		0.3144 (2.642)	1275.0729 (281.896)
Fedora-openssl	2018/04/12-2018/04/12	1	1 (1)		17 (17 - 17)		0.0149 (0.031)	-
Fedora-openssl 1.1.x-x86-64	2018/04/12-2018/04/18	21	3 (3)		29 (17 - 29)		0.012 (0.027)	1315.1925 (293.81)
address-sanitizer	2018/04/18-2018/04/18	1	13 (1)		29 (29 - 29)		0.011 (0.032)	-
centos7-openssl.1.0.x-x86-64	2018/04/18-2020/08/13	130	1 (1)		26 (17 - 25)		0.0333 (0.1)	969.0806 (457.799)
centos7-openssl.1.0.x-x86-64	2018/09/10-2021/01/20	244	1 (1)		59 (24 - 59)		0.0483 (0.168)	857.5936 (530.317)
fedora-libcrypt-x86-64	2018/06/27-2020/08/13	112	21 (17)		41 (31 - 41)		0.0632 (0.175)	1023.4279 (453.763)
fedora-libcrypt-x86-64	2018/09/10-2021/01/20	244	19 (16)		59 (40 - 59)		0.067 (0.204)	878.1499 (528.629)
fedora-mbedtls-x86-64	2018/06/27-2020/08/13	110	17 (7)		40 (30 - 40)		0.0556 (0.16)	983.861 (458.57)
fedora-mbedtls-x86-64	2018/09/10-2021/01/20	243	7 (4)		58 (39 - 58)		0.0567 (0.178)	863.7415 (527.787)
fedora-openssl.1.1.x-x86-64	2018/04/18-2020/08/13	126	15 (14)		41 (29 - 41)		0.0491 (0.132)	997.41 (449.877)
fedora-openssl.1.1.x-x86-64-release	2018/08/20-2019/03/13	56	4 (4)		41 (38 - 41)		0.0486 (0.126)	943.0119 (452.237)
fedora-openssl.1.1.x-x86-64	2018/09/10-2021/01/20	244	9 (6)		62 (40 - 62)		0.0609 (0.199)	862.0144 (529.794)
fedora-openssl.1.1.x-x86-64-fips	2019/06/13-2021/01/20	136	25 (22)		59 (55 - 59)		0.078 (0.295)	892.5534 (528.33)
fedora-openssl.1.1.x-x86-64-minimal	2018/12/13-2020/09/10	142	5 (2)		45 (40 - 45)		0.0465 (0.151)	905.5778 (535.623)
fedora-undefined-sanitizer	2018/04/18-2021/01/20	369	21 (17)		59 (29 - 57)		0.0627 (0.192)	926.7945 (513.759)
freebsd-x86-64	2018/09/10-2020/09/16	221	10 (3)		33 (24 - 33)		0.0338 (0.1)	862.8518 (539.28)
mingw32	2018/06/27-2020/08/13	88	3 (3)		16 (10 - 16)		0.093 (0.283)	947.305 (481.356)
mingw64	2018/06/27-2020/08/13	88	3 (3)		16 (10 - 16)		0.0696 (0.231)	952.9023 (476.915)
pages	2018/04/18-2018/04/18	3	18 (3)		29 (29 - 29)		0.0088 (0.022)	1434.4083 (0.608)
tumbleweed-openssl.1.1.x-x86-64	2018/05/30-2020/08/13	105	25 (9)		41 (29 - 41)		0.0568 (0.15)	959.523 (481.676)
tumbleweed-openssl.1.1.x-x86-64-release	2018/08/20-2019/03/13	56	1 (1)		41 (38 - 41)		0.06 (0.156)	958.8467 (447.542)
tumbleweed-openssl.1.1.x-x86-64-clang	2018/09/10-2021/01/20	245	29 (11)		59 (38 - 59)		0.0616 (0.192)	868.0569 (527.138)
tumbleweed-openssl.1.1.x-x86-64-gcc	2018/09/10-2021/01/20	241	28 (10)		59 (38 - 59)		0.0617 (0.192)	859.3641 (527.638)
tumbleweed-openssl.1.1.x-x86-64-gcc7	2018/09/10-2021/01/20	244	28 (11)		59 (38 - 59)		0.0623 (0.194)	865.4882 (528.868)
tumbleweed-undefined-sanitizer	2018/05/30-2020/09/10	320	37 (15)		56 (29 - 56)		0.0702 (0.195)	946.4755 (516.447)
ubuntu-openssl.1.1.x-x86-64	2019/12/23-2021/01/20	64	12 (7)		59 (57 - 59)		0.06 (0.197)	789.7291 (522.19)
undefined-sanitizer	2018/04/18-2018/04/18	4	29 (2)		29 (29 - 29)		0.0162 (0.043)	1430.6278 (1.341)
visualstudio-x86	2018/11/30-2021/01/20	177	22 (19)		18 (16 - 18)		0.0041 (0.006)	875.6071 (530.431)
visualstudio-x86-64	2018/11/30-2021/01/20	177	31 (27)		18 (16 - 18)		0.0048 (0.009)	881.6806 (528.593)

Random Approach: In each commit, this approach prioritizes the test case set available at random. During the prioritization process, neither the knowledge nor learning from previous prioritization are considered. Thus, the approach performs only exploration.

Deterministic approach: To determine the optimal solution (prioritization) for each commit associated with a system, the test case execution, as well as the failure results, are known a priori. Algorithm 1 presents the procedure adopted by the deterministic approach.

Algorithm 1: Deterministic Algorithm to Find Optimal Solutions

forall commit c in Target System **do**

$T_c \leftarrow$ Test Case set available from system in the current commit;

$A_c \leftarrow$ Total time spent to run T_c ;

$TB_c \leftarrow$ Time Budget (10%, 50%, or 80% from A_c);

$T'_c \leftarrow T_c$ ordered by number of failures (descending) and duration (ascending);

 Evaluate T'_c considering TB_c (e.g. NAPFD);

end







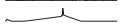



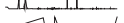


















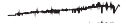
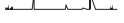




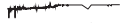
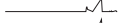

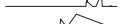

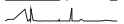











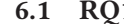





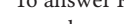

We identify the test set available T_c and the original time A_c spent to run such set in each commit. After, we define a time budget to run the tests. As aforementioned, we evaluated three time budgets (TB_c), i.e., 10%, 50%, and 80% of the execution time of the overall test set T_c available. Finally, we sort the test case by the number of failures in descending order, and test case duration in ascending order. This sorter allows evaluating the prioritized test set through different metrics, such as NAPFD.

The results were obtained from a total of 17,745 executions. From this, 9,555 are from LIBSSH system and 8,190 from DUNE system. We performed 30 independent executions for each approach (FRRMAB, ANN, and Random) in each variant (34 and 29 variants, respectively), strategy (two strategies), and time budget (three time budgets). The deterministic approach was executed only once for both systems. As mentioned before, in VTS strategy each variant is tested separately, whilst in WTS strategy all variants are executed as a unique system. All the experiments were performed on an Intel® Xeon® E5-2640 v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS.

5.4 Statistical Analysis

To evaluate a pair of performances in the same variant, we used Mann-Whitney [23] statistical test with a confidence level of 95%.

Table 2: System Information - DUNE

Name	Period	Builds	Faults	Faults/Cycle	Tests	Volatility	Duration	Interval
Total	2016/07/07-2021/01/23	2186	3094 (1010)		293 (1 - 134)		0.0236 (0.104)	1216.754 (380.477)
debian-11-gcc-9-17-downstream	2021/01/18-2021/01/18	1	14 (1)		70 (70 - 70)		0.0145 (0.015)	-
debian-11-gcc-9-17-downstream-dune-grid	2021/01/19-2021/01/19	1	14 (1)		70 (70 - 70)		0.0146 (0.017)	-
debian-11-gcc-9-17-python	2020/03/28-2021/01/20	485	747 (485)		121 (110 - 118)		0.025 (0.157)	1150.8498 (396.859)
debian_10 gcc-c_17	2017/08/02-2018/08/21	527	93 (20)		139 (32 - 120)		0.0123 (0.065)	1154.8288 (417.893)
debian_10 clang-6-libcpp-17	2018/06/29-2018/12/05	88	93 (11)		114 (15 - 112)		0.0368 (0.097)	1147.0738 (462.165)
debian_10 clang-7-libcpp-17	2018/12/05-2021/01/20	1029	1417 (811)		144 (98 - 120)		0.0131 (0.052)	1152.6743 (401.464)
debian_10 gcc-7-14-expensive	2018/06/28-2020/12/14	570	342 (36)		147 (109 - 134)		0.0139 (0.058)	1141.2207 (421.781)
debian_10 gcc-7-17	2020/03/03-2021/01/20	516	194 (27)		126 (108 - 120)		0.0132 (0.029)	1138.9429 (404.074)
debian_10 gcc-7-17-expensive	2020/03/17-2021/01/20	503	290 (79)		135 (115 - 128)		0.0132 (0.033)	1143.1528 (399.122)
debian_10 gcc-8-noassert-17	2018/06/29-2021/01/20	1104	641 (86)		144 (28 - 120)		0.0129 (0.041)	1148.3567 (407.28)
debian_11 gcc-10-20	2020/07/24-2021/01/20	290	193 (95)		125 (110 - 120)		0.0123 (0.033)	1134.5357 (394.774)
debian_11 gcc-9-20	2019/11/11-2021/01/20	707	489 (118)		130 (106 - 120)		0.0079 (0.02)	1152.2548 (398.179)
debian_9 gcc-8-clang	2016/07/11-2016/11/13	112	81 (73)		68 (57 - 67)		0.0003 (0.001)	1055.3145 (463.64)
debian_8 gcc	2016/07/07-2018/06/29	589	113 (103)		92 (57 - 85)		0.0012 (0.004)	1100.6741 (438.61)
debian_8-backports-clang	2016/10/18-2018/06/29	359	9 (9)		91 (64 - 83)		0.0011 (0.003)	1105.6547 (435.358)
debian_9 clang-3.8-14	2018/06/28-2020/12/14	615	1021 (295)		145 (11 - 118)		0.0123 (0.019)	1151.5037 (414.411)
debian_9 gcc-6-14	2018/06/29-2020/12/14	614	866 (296)		144 (16 - 118)		0.0111 (0.023)	1152.7506 (414.322)
debian_9 clang	2016/07/11-2018/08/21	1012	333 (148)		143 (56 - 120)		0.0095 (0.038)	1135.7676 (427.604)
debian_9 gcc	2016/07/07-2018/08/21	1021	259 (147)		141 (20 - 117)		0.0095 (0.039)	1134.5392 (427.435)
ubuntu-20.04-gcc-9-17-python	2020/09/29-2020/09/29	2	2 (2)		112 (112 - 112)		0.0208 (0.125)	1375.15 (0.0)
ubuntu_16.04 clang-3.8-14	2018/06/29-2020/06/26	44	223 (21)		103 (17 - 102)		0.0086 (0.014)	1198.1442 (403.978)
ubuntu_16.04 gcc-5-14	2018/06/28-2020/12/14	614	970 (297)		147 (69 - 118)		0.0124 (0.023)	1153.3314 (413.792)
ubuntu_16.04 clang	2016/10/18-2018/08/21	885	142 (64)		141 (9 - 120)		0.0107 (0.028)	1135.5978 (427.863)
ubuntu_16.04 gcc	2016/10/18-2018/08/21	894	169 (69)		141 (37 - 117)		0.0101 (0.037)	1139.1877 (423.91)
ubuntu_18.04 clang-5-17	2020/05/28-2021/01/20	368	29 (21)		122 (109 - 120)		0.011 (0.018)	1141.8334 (396.813)
ubuntu_18.04 clang-6-17	2018/06/28-2021/01/20	1121	582 (77)		148 (77 - 120)		0.0116 (0.016)	1151.4211 (405.942)
ubuntu_20.04 clang-10-20	2020/05/28-2021/01/20	367	532 (367)		122 (109 - 120)		0.0122 (0.025)	1141.0444 (399.455)
ubuntu_20.04 gcc-10-20	2020/10/03-2021/01/20	243	43 (37)		125 (110 - 120)		0.012 (0.027)	1145.5798 (385.416)
ubuntu_20.04 gcc-9-20	2020/10/03-2021/01/20	242	19 (13)		125 (110 - 120)		0.0107 (0.026)	1146.748 (378.89)

Furthermore, we used the Vargha and Delaney's \hat{A}_{12} [49] to calculate the difference between two groups. \hat{A}_{12} evaluate the probability of a value, taken randomly from the first sample, is higher than a value taken randomly from the second sample. This metric provides a magnitude scale: (i) *Negligible*, represents a very small difference among the values and usually does not yield statistical difference; (ii) *Small* and *Medium*, represent small and medium differences among the values, and may yield statistical differences; and (iii) *Large* magnitude represents a significantly large difference that usually can be seen in the numbers without much effort.

6 RESULTS AND ANALYSIS

In this section we present and analyze the results obtained. Due to the large number of variants, and to improve visualization in the tables, we named each variant as L_i , \forall variant $i \in$ LIBSSH system, and D_i , \forall variant $i \in$ DUNE system. The i th value represents the position of the variant i in Tables 1 and 2 for the systems LIBSSH and DUNE, respectively. For instance, L_1 represents CentOS7-openssl, the LIBSSH variant described in the first line of Table 1.

The results are presented in a summarized way in order to support our analysis and findings. The values of NAPFD, NTR, PT, and RMSE obtained for each approach for both systems, including statistical test results and effect size, can be found in our supplementary material (see [36]).

6.1 RQ1: Performance of the strategies using COLEMAN and random approach

To answer RQ1, we compare the results from both strategies using a random prioritization to the results obtained using COLEMAN with the FRRMAB policy. Such an analysis encompasses the three time budgets. To compare the performance, we used NAPFD as the main quality indicator. Tables 3 and 4 present the values obtained for the LIBSSH and DUNE systems, respectively.

In such tables, for each approach, strategy, and time budget, we present in the second column the variants in which the approaches are statistically equivalent. In the other columns, we present the variants in which an approach was better than another with a statistical difference. For instance, in Table 3 (LIBSSH), time budget 10%, the FRRMAB approach using VTS strategy has the best performance in 20 variants (out of 34, 59% of the cases), Random is the best in five variants out of 34 (15%). The approaches are equivalent in nine variants.

When compared with the random approach, COLEMAN using FRRMAB is the best approach with a statistical difference in most cases in both systems. Overall, considering both strategies, the three time budgets, and 34 variants for the LIBSSH system ($2 \times 3 \times 34 = 204$ cases) and 29 for the DUNE system ($2 \times 3 \times 29 = 174$ cases), COLEMAN is the best approach with a statistical difference in 175 cases out of 204 ($\approx 86\%$) and in 167 out of 174 (96%), respectively.

In the DUNE system (Table 4), random prioritization does not present a good performance. There is no case where it has the best performance and there are seven (4% of the cases) where it reaches

Table 3: FRRMAB x Random - NAPFD values considering VTS and WTS strategies for the LIBSSH system

Strategy	Equivalent	Random	FRRMAB
Time Budget 10%			
VTS	9 (26%) {L ₁ ,L ₅ -L ₇ , L ₉ ,L ₁₀ ,L ₁₃ , L ₂₇ ,L ₃₄ }	5 (15%) {L ₁₇ ,L ₂₀ , L ₂₈ -L ₃₀ }	20 (59%) {L ₂ -L ₄ ,L ₈ ,L ₁₁ , L ₁₂ ,L ₁₄ -L ₁₆ , L ₁₈ ,L ₁₉ , L ₂₁ -L ₂₆ ,L ₃₁ -L ₃₃ }
WTS	5 (15%) {L ₄ ,L ₉ ,L ₁₀ , L ₁₈ ,L ₂₆ }	0 (0%)	29 (85%) {L ₁ -L ₃ , L ₅ -L ₈ ,L ₁₁ -L ₁₇ , L ₁₉ -L ₂₅ ,L ₂₇ -L ₃₄ }
Time Budget 50%			
VTS	2 (6%) {L ₆ ,L ₈ }	2 (6%) {L ₁ ,L ₂₈ }	30 (88%) {L ₂ -L ₅ ,L ₇ ,L ₉ -L ₂₇ , L ₂₉ -L ₃₄ }
WTS	1 (3%) {L ₄ }	0 (0%)	33 (97%) {L ₁ -L ₃ ,L ₅ -L ₃₄ }
Time Budget 80%			
VTS	3 (9%) {L ₁ ,L ₆ ,L ₈ }	1 (3%) {L ₉ }	30 (88%) {L ₃ -L ₅ ,L ₈ ,L ₁₀ -L ₃₄ }
WTS	1 (3%) {L ₉ }	0 (0%)	33 (97%) {L ₁ -L ₈ ,L ₁₀ -L ₃₄ }

Table 4: FRRMAB x Random - NAPFD values considering VTS and WTS strategies for the DUNE system.

Strategy	Equivalent	Random	FRRMAB
Time Budget 10%			
VTS	3 (10%) {D ₁ ,D ₂ ,D ₂₀ }	0 (0%)	26 (90%) {D ₃ -D ₁₉ ,D ₂₁ -D ₂₉ }
WTS	0 (0%)	0 (0%)	29 (100%) {D ₁ -D ₂₉ }
Time Budget 50%			
VTS	2 (7%) {D ₁ ,D ₂ }	0 (0%)	27 (93%) {D ₃ -D ₂₉ }
WTS	0 (0%)	0 (0%)	29 (100%) {D ₁ -D ₂₉ }
Time Budget 80%			
VTS	2 (7%) {D ₁ ,D ₂ }	0 (0%)	27 (93%) {D ₃ -D ₂₉ }
WTS	0 (0%)	0 (0%)	29 (100%) {D ₁ -D ₂₉ }

equivalent results. On the other hand, in the LIBSSH system (Table 3), Random is the best in eight cases (4%), and reaches statistical equivalence results in 21 cases (10%). The equivalence is obtained only when random is applied with VTS. We also observe that for both systems random never reaches the best performance with WTS. With COLEMAN seems to happen the opposite. Thus, it seems that finding good prioritizations in the WTS strategy is harder than in the VTS strategy.

Finding 1

Random presents better performance using the WTS strategy in both systems.

We observed that when there is a statistical difference, the effect size tends to be large. Furthermore, cases with statistical equivalence appear across the time budgets. In these few cases, the variants have few failures (most of them have only one failure), and clearly, COLEMAN has no significant performance.

Finding 2

COLEMAN and Random have equivalent performance across the time budgets in cases with few failures (most of them only one failure).

We also can see that COLEMAN outperforms the random prioritization, independently of the strategy used. If we use the VTS strategy, FRRMAB is the best one in 80 out of 102 cases (34 variants \times 3 budgets) (78%) and presents equivalent results in 14 (13%) in LIBSSH. In the DUNE system, it is the best in 80 out of 87 cases (considering 29 variants \times 3) (92%) and equivalent in seven (8%). If we use WTS, FRRMAB is the best in 95 cases (out of 102) (93%) for the LIBSSH system, and in all cases for the DUNE system. It is statistically equivalent in seven (7%) cases for the LIBSSH system, and is not the worst in any case.

Finding 3

COLEMAN presents better performance than Random in both systems, independently of the strategy used. Considering both systems (189 cases), COLEMAN obtained the best results (with a statistical difference) in 85% of the cases and equivalent in 11% for VTS strategy. For WTS it is the best in 96% of the cases and equivalent in 3%.

The COLEMAN's worst performance is in few variants in both systems and using VTS strategy. Considering the LIBSSH system, the worst performance is obtained for the variants CentOS7-openssl (L₁), Fedora-openssl (L₆), address-sanitizer (L₈), and pages (L₂₄). These variants share some important characteristics that hamper a good prioritization: one to three commits as historical test data and low failing cycles. Besides, in L₁ and L₆, only one failure can be revealed.

Finding 4

COLEMAN presents the worst performance for variants with few CI Cycles and a small number of failing cycles.

In a more in-depth analysis, we can observe other relevant aspects, for instance, in the variant address-sanitizer (L₈). In this variant, we have only one commit and 13 test cases that fail from 29 available. Before this variant, only nine variants are defined, and 13 commits have some test case which failed. From these commits, the failed test cases are scattered between the variants. From the test set available for this variant, only 11 test cases have historical failure data. Some take more time (duration) to execute from the available test cases than others, and among them, the failed test cases. This shows that sometimes we face test cases that fail but spend much time executing, which hampers a reasonable prioritization.

Finding 5

Test cases that fail but spend much time executing hamper a good prioritization.

On the other hand, we observed the worst performance in the DUNE system for the variants `debian-11-gcc-9-17-downstream` (D_1), `debian-11-gcc-9-17-downstream-dune-grid` (D_2), and `ubuntu-20_04-gcc-9-17-python` (D_{20}). Such variants share the same characteristics identified in the LIBSSH system variants, and a relevant (big) number of test cases to be prioritized.

The biggest difference between the approaches can be observed in the variant `ubuntu-openssl_1.1.x-x86_64` (L_{31}) in the LIBSSH system. Although this variant has few commits, there is a high test case volatility, mainly in two commits in which failures occur¹⁰. Before these two commits, only one commit failed. Considering the DUNE system, we observed biggest difference in the variants `debian-11-gcc-9-17-python` (D_3), `debian_10 clang-7-libcpp-17` (D_6), and `ubuntu_20_04 gcc-10-20` (D_{28}). Such variants have many historical test data, high test case volatility, and failing commits. When we observe the characteristics of each variant, we identify the reason behind the difference. The variants D_3 and D_6 have few peaks of failures (one and four, respectively), and the same test case fails in all commits. Consequently, in such a scenario, our learning-based approach obtains better results than random prioritization. On the other hand, the variant D_{28} has a peak of failures but long periods without failures, and this allows better performance for our approach based on historical test data.

Finding 6

COLEMAN deals adequately with a high test case volatility, and obtains good prioritizations even when there is peaks or long periods without failures.

Some of these findings are similar to the ones about COLEMAN performance considering systems in general [37, 38]. They corroborate to validate those results in the particular context of HCSs.

Answer for RQ1

Independently of the time budget, we can then conclude that both strategies have the best performance with COLEMAN against random, for both systems. COLEMAN reaches the best results or statistically equivalent ones in the great majority of the cases (96%) for LIBSSH system and all cases (100%) for DUNE system.

¹⁰See supplementary material about characteristics of this variant. Available in [36].

6.2 RQ2: Performance of the strategies using COLEMAN and RETECS

To answer RQ2, we compare the results obtained from both strategies for RETECS using an ANN against COLEMAN using FRRMAB. For a fair comparison, we conducted the same analysis performed in RQ1, but based on Tables 5 and 6.

Table 5: FRRMAB x ANN - NAPFD values considering VTS and WTS strategies for the LIBSSH system.

Strategy	Equivalent	ANN	FRRMAB
Time Budget 10%			
VTS	6 (18%) {L ₄ ,L ₈ ,L ₉ , L ₁₆ ,L ₂₆ , L ₃₄ }	3 (9%) {L ₁₉ ,L ₂₈ , L ₂₉ }	24 (71%) {L ₁ -L ₃ ,L ₅ -L ₇ , L ₁₀ -L ₁₅ ,L ₁₇ , L ₁₈ ,L ₂₀ -L ₂₅ , L ₂₇ ,L ₃₀ -L ₃₃ }
WTS	4 (12%) {L ₄ ,L ₉ ,L ₁₀ , L ₂₆ }	0 (0%)	30 (88%) {L ₁ -L ₃ , L ₅ -L ₈ ,L ₁₁ -L ₂₅ , L ₂₇ -L ₃₄ }
Time Budget 50%			
VTS	3 (9%) {L ₁ ,L ₆ ,L ₈ }	0 (0%)	31 (91%) {L ₂ -L ₅ ,L ₇ ,L ₉ -L ₃₄ }
WTS	0 (0%)	0 (0%)	34 (100%) {L ₁ -L ₃₄ }
Time Budget 80%			
VTS	2 (6%) {L ₁ ,L ₆ }	1 (3%) {L ₉ }	30 (88%) {L ₂ -L ₅ ,L ₇ ,L ₈ ,L ₁₀ -L ₃₄ }
WTS	0 (0%)	0 (0%)	34 (100%) {L ₁ -L ₃₄ }

Table 6: FRRMAB x ANN - NAPFD values considering VTS and WTS strategies for the DUNE system.

Strategy	Equivalent	ANN	FRRMAB
Time Budget 10%			
VTS	2 (7%) {D ₂ ,D ₂₀ }	0 (0%)	27 (93%) {D ₁ ,D ₃ -D ₁₉ ,D ₂₁ -D ₂₉ }
WTS	0 (0%)	1 (3%) {D ₁ }	28 (97%) {D ₂ -D ₂₉ }
Time Budget 50%			
VTS	1 (3%) {D ₁ }	1 (0%) {D ₂ }	27 (97%) {D ₃ -D ₂₉ }
WTS	0 (0%)	1 (3%) {D ₁ }	28 (97%) {D ₂ -D ₂₉ }
Time Budget 80%			
VTS	2 (7%) {D ₁ ,D ₂ }	0 (0%)	27 (93%) {D ₃ -D ₂₉ }
WTS	0 (0%)	1 (3%) {D ₁ }	28 (97%) {D ₂ -D ₂₉ }

In comparison with RETECS, COLEMAN is the best in the great majority of the cases. Again, considering both strategies and systems, and the three budgets, COLEMAN is the best approach with statistical difference, in 183 cases out of 204 (89%) and in 165 out of 174 (95%), respectively for the systems DUNE and LIBSSH. They are equivalent in 17 cases (8%) for LIBSSH, and in five cases (2%) for DUNE. We also observed that when there is a statistical difference, the effect size tends to be large.

We can observe that RETECS results when compared with COLEMAN are similar to those obtained by random: RETECS does not present good performance for DUNE, and performed best or equivalent with VTS and with a more restrictive budget. The sets of variants corresponding to these cases have intersection with the sets of variants corresponding to the cases where random performed best or equivalent to COLEMAN. We can mention the variants L_1 , L_5 , L_6 , L_8 , L_9 , L_{28} , L_{34} and D_1 , D_2 . Some of them are the hard cases for COLEMAN as discussed in previous subsection.

We also can see that COLEMAN outperforms RETECS independently of the strategy used. If we use the VTS strategy, FRRMAB is the best one in 85 out of 102 cases (34 variants \times 3 budgets) (83%) and presents equivalent results in 13 (12.7%) in LIBSSH. In the DUNE

system, it is the best in 81 out of 87 cases (considering 29 variants \times 3) (93%) and equivalent in five (6%). If we use WTS, FRRMAB is the best in 98 cases (out of 102) (96%) for the LIBSSH system, and in 84 cases (96%) for the DUNE system. It is statistically equivalent in four (4%) cases for the LIBSSH system, and there is no case of equivalence for DUNE.

Finding 7

COLEMAN presents better performance than RETECS in both systems, independently of the strategy used. Considering both systems (189 cases), COLEMAN obtained the best results (with statistical difference) in 88% of the cases and equivalent in 10% for VTS strategy. For WTS it is the best in 96% of the cases and equivalent in 2%.

Answer for RQ2

We can conclude that independently of the time budget, both strategies have the best performance with COLEMAN compared with RETECS. COLEMAN obtained the best results or statistically equivalent ones in 98% of the cases for both systems.

6.3 RQ3: How far the solutions found by all approaches are from optimal solutions.

To answer this question, we use in the comparison a prioritization generated deterministically by using a priori knowledge about the test results. RMSE metric is used to compare the accuracy of the approaches, concerning NAPFD values. This allows us to observe how distant the solutions produced are from optimal solutions. Although NAPFD metric ranges between 0 and 1, a time budget hampers to obtain its maximum value. In this way, the deterministic approach shows the maximum values possible to reach in each evaluation.

As expected and can be viewed in the tables of our repository (see [36]), the deterministic approach presents the best NAPFD values with a statistical difference in almost all systems, budgets, and strategies that has, in most cases, a *large* magnitude. Although this difference appears in the great majority of the cases, in some cases, we observe statistical equivalence between COLEMAN using the WTS strategy and the deterministic approach. Moreover, such a strategy using COLEMAN obtained the best values than the deterministic approach in the variants for the LIBSSH system: address-sanitizer (L_8) for all time budgets, and undefined-sanitizer (L_{32}) for the budgets of 50% and 80%.

The variants address-sanitizer (L_8) and undefined-sanitizer (L_{32}) from the LIBSSH system show a drawback for the use of WTS. Such a strategy defines a unique prioritized test set to be used across the variants but the order defined can not be valid in all variants. For instance, considering these variants, the deterministic approach using the WTS strategy defined an optimal order that contains a test case that fails in

undefined-sanitizer (L_{32}) but not in address-sanitizer (L_8). On the other hand, COLEMAN mitigates this by considering the knowledge from previous prioritizations. The difference was notice in these variants because they are the hardest variants to prioritize (lowest NAPFD values found). These variants have less than four commits, and prioritizing individually each variant thought a deterministic way the maximum NAPFD values possible are 0.7759 and 0.8750, respectively.

Finding 8

The prioritization order defined by the WTS strategy can not be the best order for all variants, because some test cases that do not fail in some variants can appear before that a failing one. The use of WTS with COLEMAN is fundamental to mitigate this, to allow considering the knowledge from previous prioritizations.

When we compare only the NAPFD results obtained by the deterministic approach in both strategies, we observe that the results are equals or very close. For instance, the variant CentOS7-openssl (L_1) presents the same NAPFD value in both strategies. In the variant visualstudio-x86_64 (L_{34}) the best NAPFD value found by the VTS strategy is 0.9978, and the WTS strategy reaches 0.9931 in the time budget of 10% and 0.9957 in the other time budgets. Another example is centos7-openssl_1_0_x-x86_64 (L_{10}), in which the maximum NAPFD value found by the VTS strategy is 1 and the WTS strategy reaches 0.9999. The facts observed show that the WTS strategy is a valid strategy to be used. However, an approach that reaches solutions close to optimal, when compared to the VTS strategy, is a challenge.

Finding 9

The NAPFD values found by the deterministic approach with both strategies are equals or very close. This suggests that prioritizing each variant individually or as a unique system has an equivalent performance. Although the WTS strategy can seem impracticable, because each variant has individual behavior and characteristics, this finding shows the opposite.

To evaluate the accuracy (RMSE) obtained by COLEMAN and the other two compared approaches, we evaluated the difference between the predicted and the observed values of NAPFD; these last ones generated by the deterministic approach. The results are available in the supplementary material (see [36]). To a better visualization, we generated Figures 7, 8, 9, and 10 that present the distribution of RMSE values found in each strategy and variant across the time budgets. In the RMSE metric, the lowest values represent better performance.

We observe that COLEMAN obtains more cases with lowest RMSE values in all time budgets, both strategies, and systems. In some cases, there is a large difference between COLEMAN and the other

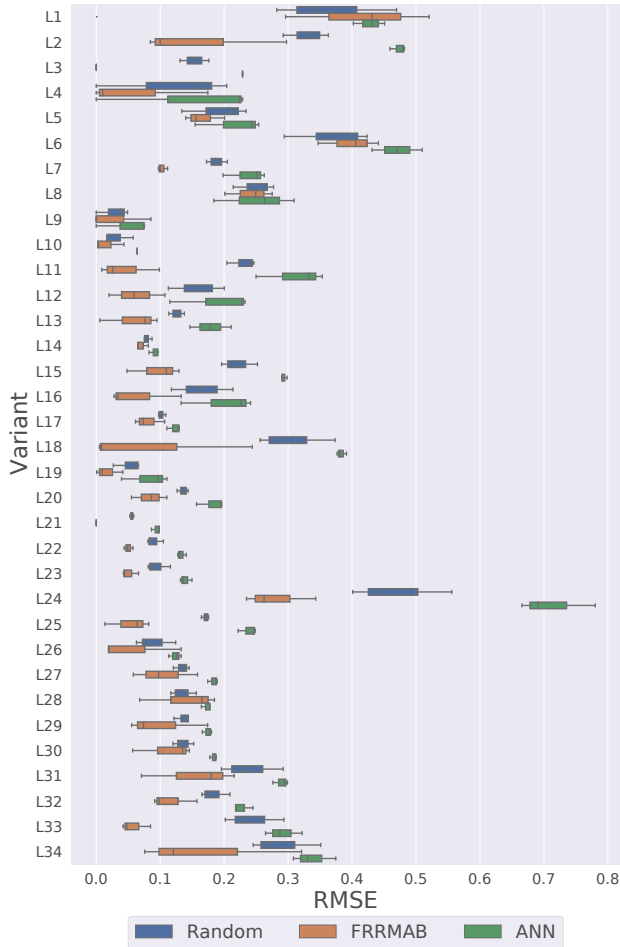


Figure 7: RMSE values found in the LIBSSH system for the VTS strategy.

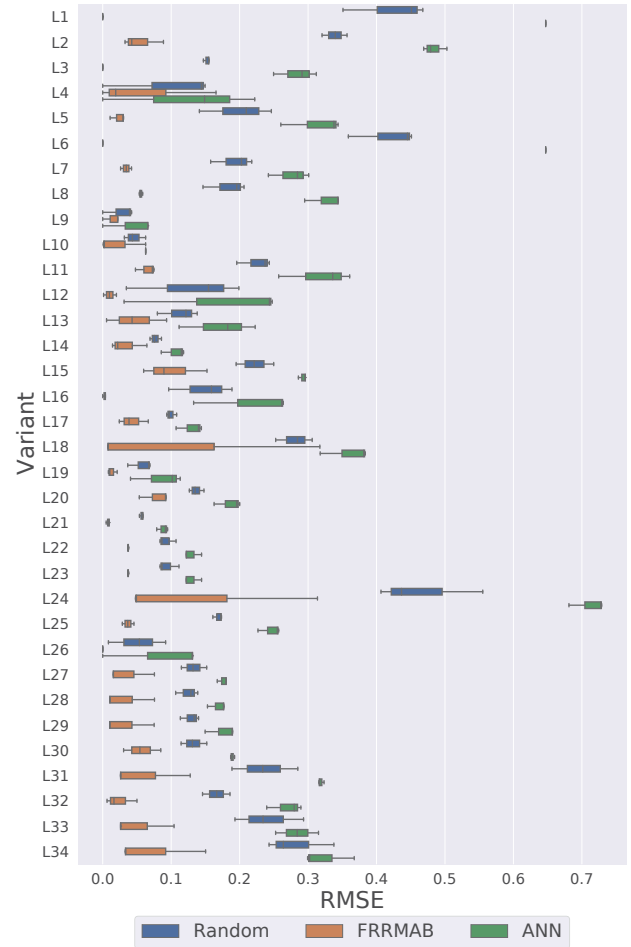


Figure 8: RMSE values found in the LIBSSH system for the WTS strategy.

approaches. Comparing the strategies using COLEMAN, WTS presents more cases near to optimal solutions (close to zero). Tables 7 and 8 present an overview of the results found. Each RMSE value has a corresponding magnitude that are presented in such tables.

As we can observe, the WTS strategy, in the LIBSSH system and time budget of 80%, reaches very near optimal solutions in 100% of the cases. Regarding reasonable solutions, when $RMSE < 0.3$ [38], WTS also presents better performance reaching 98% of the cases in both systems. On the other hand, VTS reaches 93% and 92% of the cases for the LIBSSH and DUNE systems, respectively.

Finding 10

COLEMAN using the WTS strategy presents better performance, providing reasonable solutions in 98% of the cases in both systems.

On the other hand, Random reaches reasonable solutions around 64% in the DUNE system. Considering the LIBSSH system, the percentage of reasonable solutions found by Random increases to 90% using the VTS strategy and 86% using the WTS strategy. RETECS has the worst performance in comparison with the deterministic approach. It reaches reasonable solutions in 45% of the cases in the DUNE system, and a better performance in LIBSSH, 78% using the VTS strategy and 72% using the WTS strategy. For RETECS and random approaches we observe a low performance for DUNE and no great differences between both strategies.

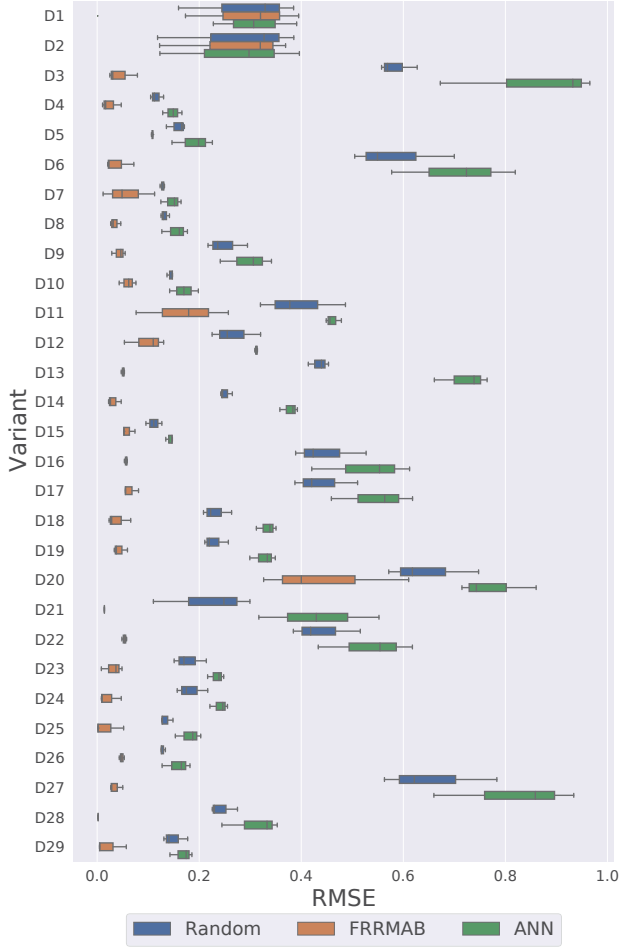


Figure 9: RMSE values found in the DUNE system for the VTS strategy.

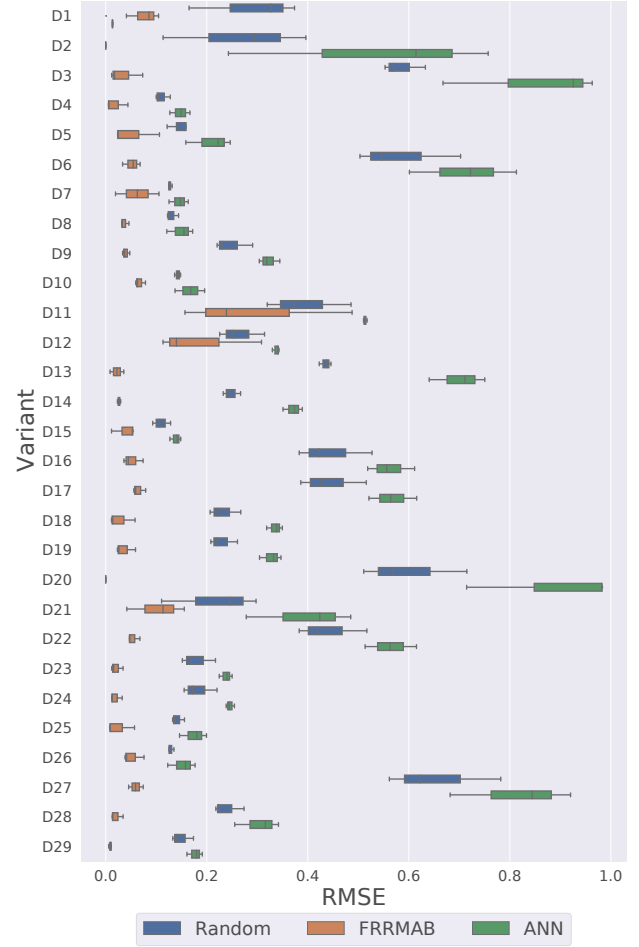


Figure 10: RMSE values found in the DUNE system for the WTS strategy.

Answer for RQ3

In comparison with the deterministic approach, the strategies using COLEMAN provide reasonable solutions in most of the cases (more than 92%) in both systems. Furthermore, the optimal solutions found by the deterministic approach using the WTS strategy are very close or equals to that ones obtained using the VTS strategy.

6.4 RQ4: Strategies Applicability

To analyze the applicability of the strategies using COLEMAN, we compute the time (PT values in seconds) spent to prioritize the test cases considering the three budgets. An overview of the results is depicted in Tables 9 and 10.

We observe that both strategies spend in all cases less than one second in both systems. The worst cases were identified using the VTS strategy. In the LIBSSH system, the VTS strategy spends

0.0396 seconds in variant address-sanitize (L_8) and time budget of 50%. On the other hand, in the DUNE system and time budget of 10%, the VTS strategy spends 0.0418 seconds in variant debian-11-gcc-9-17-downstream (D_1). Besides, we can observe that the time budget has no influence in the prioritization time.

Finding 11

The time spent to execute the strategies using COLEMAN has a low computational cost. Furthermore, the prioritization time is not influenced by the time budget and is negligible (less than 0.042 seconds).

Analyzing individually each system based on PT, we observe that in the LIBSSH system the VTS strategy is the best strategy, with a statistical difference, for the time budget of 10% in $\approx 71\%$ (24 out of 34) of the cases. WTS is the best in $\approx 15\%$ (5). Considering

Table 7: RMSE magnitudes for NAPFD values found for LIBSSH system.

Scale	VTS			WTS		
	Random	FRRMAB	ANN	Random	FRRMAB	ANN
	Time Budget 10%					
very near	18(53%)	22(65%)	13(38%)	19(56%)	30(88%)	14(41%)
near	7(21%)	6(18%)	11(32%)	6(18%)	2(6%)	5(15%)
reasonable	4(12%)	3(9%)	3(9%)	3(9%)	0(0%)	7(21%)
far	1(3%)	3(9%)	1(3%)	2(6%)	2(6%)	3(9%)
very far	4(12%)	0(0%)	6(18%)	4(12%)	0(0%)	5(15%)
	Time Budget 50%					
very near	15(44%)	26(76%)	9(26%)	16(47%)	33(97%)	10(29%)
near	10(29%)	4(12%)	9(26%)	9(26%)	1(3%)	6(18%)
reasonable	6(18%)	2(6%)	9(26%)	5(15%)	0(0%)	9(26%)
far	1(3%)	0(0%)	2(6%)	1(3%)	0(0%)	4(12%)
very far	2(6%)	2(6%)	5(15%)	3(9%)	0(0%)	5(15%)
	Time Budget 80%					
very near	15(44%)	29(85%)	9(26%)	17(50%)	34(100%)	9(26%)
near	10(29%)	1(3%)	10(29%)	9(26%)	0(0%)	7(21%)
reasonable	6(18%)	2(6%)	7(21%)	4(12%)	0(0%)	6(18%)
far	0(0%)	0(0%)	2(6%)	1(3%)	0(0%)	6(18%)
very far	3(9%)	2(6%)	6(18%)	3(9%)	0(0%)	6(18%)
Reasonable	91(90%)	95(93%)	80(78%)	88(86%)	100(98%)	73(72%)

Table 8: RMSE magnitudes for NAPFD values found for the Dune system.

Scale	VTS			WTS		
	Random	FRRMAB	ANN	Random	FRRMAB	ANN
	Time Budget 10%					
very near	10(34%)	26(90%)	9(31%)	9(31%)	27(93%)	8(28%)
near	4(14%)	1(3%)	4(14%)	5(17%)	0(0%)	3(10%)
reasonable	5(17%)	1(3%)	3(10%)	5(17%)	0(0%)	4(14%)
far	1(3%)	0(0%)	3(10%)	1(3%)	1(3%)	4(14%)
very far	9(31%)	1(3%)	10(34%)	9(31%)	1(3%)	10(34%)
	Time Budget 50%					
very near	8(28%)	25(86%)	2(7%)	8(28%)	28(97%)	4(14%)
near	6(21%)	1(3%)	7(24%)	6(21%)	0(0%)	6(21%)
reasonable	4(14%)	0(0%)	3(10%)	5(17%)	1(3%)	2(7%)
far	2(7%)	2(7%)	6(21%)	1(3%)	0(0%)	5(17%)
very far	9(31%)	1(3%)	11(38%)	9(31%)	0(0%)	12(41%)
	Time Budget 80%					
very near	8(28%)	26(90%)	1(3%)	8(28%)	27(93%)	2(7%)
near	8(28%)	0(0%)	8(28%)	8(28%)	2(7%)	7(24%)
reasonable	2(7%)	0(0%)	2(7%)	2(7%)	0(0%)	3(10%)
far	1(3%)	1(3%)	3(10%)	1(3%)	0(0%)	5(17%)
very far	10(34%)	2(8%)	15(52%)	10(34%)	0(0%)	12(41%)
Reasonable	55(63%)	80(92%)	39(45%)	56(64%)	85(98%)	39(45%)

the time budgets of 50% and 80%, the results are similar, VTS is the best in $\approx 74\%$ (25) of the cases in both time budgets, and WTS in 18% (6) and 15% (5), respectively.

Regarding the DUNE system, the VTS strategy is also the best strategy. The VTS strategy is the best in 90% of the cases in all time budgets, and the WTS strategy obtains the best performance in 7% of the cases for time budget of 10%, 3% in the budget of 50%, and 10% in the budget of 80%. The strategies are equivalent only in the time budgets of 10% and 50%, obtaining equivalence in 3% and 7% of the cases, respectively.

Table 9: Summary comparison VTS x WTS about PT values in the LIBSSH system.

	Time Budget 10%		Time Budget 50%		Time Budget 80%	
Equivalent	5 (15%)	{L ₂ ,L ₃ ,L ₅ , L ₇ ,L ₁₅ }	3 (9%)	{L ₃ ,L ₅ ,L ₇ }	4 (12%)	{L ₂ ,L ₃ ,L ₅ ,L ₇ }
VTS	24 (71%)	{L ₄ ,L ₉ -L ₁₄ , L ₁₆ -L ₂₃ , L ₂₅ -L ₃₁ , L ₃₃ ,L ₃₄ }	25 (74%)	{L ₄ ,L ₉ -L ₂₃ , L ₂₅ -L ₃₁ , L ₃₃ ,L ₃₄ }	25 (74%)	{L ₄ ,L ₉ -L ₂₃ , L ₂₅ -L ₃₁ , L ₃₃ ,L ₃₄ }
Worst Time	0.0391	{L ₁ }	0.0396	{L ₈ }	0.0358	{L ₈ }
WTS	5 (15%)	{L ₁ ,L ₆ ,L ₈ L ₂₄ ,L ₃₂ }	6 (18%)	{L ₁ ,L ₂ ,L ₆ ,L ₈ L ₂₄ ,L ₃₂ }	5 (15%)	{L ₁ ,L ₆ ,L ₈ L ₂₄ ,L ₃₂ }
Worst Time	0.0271	{L ₃₁ }	0.0273	{L ₃₁ }	0.0273	{L ₃₁ }

Table 10: Summary comparison VTS x WTS about PT values in the DUNE system.

	Time Budget 10%		Time Budget 50%		Time Budget 80%	
Equivalent	1 (3%)	{D ₂₀ }	2 (7%)	{D ₁ ,D ₂₀ }	0 (0%)	
VTS	26 (90%)	{D ₃ -D ₁₉ , D ₂₁ -D ₂₉ }	26 (90%)	{D ₃ -D ₁₉ , D ₂₁ -D ₂₉ }	26 (90%)	{D ₃ -D ₁₉ , D ₂₁ -D ₂₉ }
Worst Time	0.0418	{D ₁ }	0.0401	{D ₁ }	0.0392	{D ₂ }
WTS	2 (7%)	{D ₁ ,D ₂ }	1 (3%)	{D ₂ }	3 (10%)	{D ₁ ,D ₂ ,D ₂₀ }
Worst Time	0.0345	{D ₁ }	0.0336	{D ₁ }	0.0338	{D ₁ }

If analyze the duration of the CI cycles and the interval between them, presented in Tables 1 and 2 for both systems, we observe that a commit is typically performed after another one is finished and with a considerable time. Considering the commit duration, the variants used in both systems do not present a situation with multiple test requests for the same variant. Then, we can conclude that both strategies using COLEMAN are applicable considering the time between commits.

Answer for RQ4

Regarding the time spent prioritizing the test cases, both VTS and WTS strategies spend less than one second to execute. In addition to this, we do not observe any impact on the time budgets. Considering the facts mentioned above, the time spent by the strategies is feasible. That is, both strategies are applicable in the CI context.

6.5 RQ5: Comparing VTS and WTS strategies

To compare both strategies using COLEMAN we analyze NAPFD and applied statistical tests. Again, for the sake of space, we present only a summary in Tables 11 and 12. Tables with the NAPFD values are detailed in our supplementary material (see [36]).

We observe that WTS is the best in the LIBSSH system, and VTS in the DUNE system. Considering the LIBSSH system, VTS obtains the best performance, with a statistical difference, in 15%, 24%, and 38% of the cases for the time budgets 10%, 50%, and 80%, respectively.

In comparison, WTS is the best in 65%, 62%, and 53%. The strategies are equivalent in 21%, 15%, and 9%. Considering all budgets and variants (102 cases), VTS is the best in 26 cases (25%), and WTS is the best in 61 cases (60%). They are statistically equivalent in 15 cases (15%).

On the other hand, in the DUNE system, WTS is closest to VTS in the time budget of 10%. VTS obtains the best performance for this budget, with a statistical difference, in 17 cases (out of 29, $\approx 59\%$). WTS is the best in only $\approx 34\%$ (10 cases). In the other time budgets, VTS is the best in 20 cases (69%) for time budget of 50%, and 22 cases (76%) for 80%. WTS is the best in 28% and 24%. The strategies are equivalent in 7% and 3% for the time budgets 10% and 50%, and they are not equivalent in 80%. Overall, VTS is the best in 59 cases (68%), WTS in 25 cases (29%), and they are equivalent in three cases (3%).

Overall, increasing the time budget, the VTS performance increases, and WTS decreases for both systems. This suggests WTS provides better results with a less restrictive situation (more time budget), and VTS the opposite. Interestingly they are equivalent in few cases. If we consider all budgets, the strategies are equivalent in 15 cases (out of 102, 15%) and only three (out of 87, 3.4%) for, respectively, LIBSSH and DUNE. This number is greater in the more restrictive scenario (budget of 10%) and decreases, increasing the budget.

Finding 12

The strategies have opposite behaviors. VTS provides better NAPFD values with more time budget, and WTS the opposite. The greater the budget, the lower is the number of equivalent cases.

Although NAPFD provides a good way to analyze the prioritization quality, it does not encompass the situation in which the tests are ended when a failure is revealed. For this, we use the NTR metric to observe the impact of the order to reduce the test duration process. The results are provided in our supplementary material (see [36]), and Tables 13 and 14 present a summary of the results.

We observe that considering the NTR values for the LIBSSH system, WTS is the best strategy in 23 cases for time budget 10%, 20 for 50%, and 15 for 80%, whilst VTS is the best, respectively in five, 11, and 16 cases. These strategies have equal performance in six, three, and three cases (out of 34). WTS reaches the maximum reduction of 99.90% in the three time budgets evaluated in the same variant L_8 (address-sanitizer). Such a variant has only one commit, and this fact shows the importance of WTS to mitigate the problem in beginning without learning. Whilst, VTS has the best reduction of 98.53% for 10% (variant L_8), 93.88% for 50% (variant pages L_{24}), and 95.23% for 80% (variant L_{24}). In this system, WTS is the best for a restrictive scenario while VTS the opposite. This observation corroborates with the results found previously.

Regarding the DUNE system, the strategies present similar behavior identified in the LIBSSH system. WTS presents a better performance in the time budget of 10%, and VTS in the other ones. The strategies are equivalent in 3% for the time budgets 10% and 50%.

Table 11: Summary comparison VTS x WTS about NAPFD values in the LIBSSH system.

	Time Budget 10%	Time Budget 50%	Time Budget 80%
Equivalent	7 (21%) {L ₃ ,L ₄ ,L ₉ , L ₂₂ -L ₂₄ ,L ₂₆ }	5 (15%) {L ₃ ,L ₁₅ ,L ₁₆ , L ₁₈ ,L ₂₀ }	3 (9%) {L ₃ ,L ₁₂ ,L ₁₆ }
VTS	5 (15%) {L ₁₀ ,L ₁₃ ,L ₁₅ , L ₁₈ ,L ₂₁ }	8 (24%) {L ₄ ,L ₉ ,L ₁₁ ,L ₁₃ , L ₁₉ ,L ₂₁ -L ₂₃ }	13 (38%) {L ₄ ,L ₁₀ ,L ₁₁ ,L ₁₃ , L ₁₅ ,L ₁₈ -L ₂₄ , L ₂₅ ,L ₃₀ }
WTS	22 (65%) {L ₁ ,L ₂ ,L ₅ -L ₈ , L ₁₁ ,L ₁₂ ,L ₁₄ ,L ₁₆ , L ₁₇ ,L ₁₉ ,L ₂₀ ,L ₂₅ , L ₂₇ -L ₃₄ }	21 (62%) {L ₁ ,L ₂ ,L ₅ -L ₈ , L ₁₀ ,L ₁₂ ,L ₁₄ , L ₁₇ ,L ₂₄ -L ₃₄ }	18 (53%) {L ₁ ,L ₂ ,L ₅ -L ₉ , L ₁₄ ,L ₁₇ ,L ₂₄ , L ₂₆ -L ₂₉ , L ₃₁ -L ₃₄ }

Table 12: Summary comparison VTS x WTS about NAPFD values in the DUNE system.

	Time Budget 10%	Time Budget 50%	Time Budget 80%
Equivalent	2 (7%) {D ₃ ,D ₁₃ }	1 (3%) {D ₁₄ }	0 (0%)
VTS	17 (59%) {D ₅ ,D ₆ ,D ₈ -D ₁₂ , D ₁₄ ,D ₁₆ ,D ₁₇ ,D ₁₉ , D ₂₁ ,D ₂₂ ,D ₂₅ -D ₂₈ }	20 (69%) {D ₃ ,D ₆ -D ₁₃ , D ₁₅ -D ₁₇ ,D ₂₁ , D ₂₂ ,D ₂₄ -D ₂₉ }	22 (76%) {D ₃ ,D ₆ -D ₁₄ ,D ₁₆ , D ₁₇ ,D ₁₉ ,D ₂₁ -D ₂₉ }
WTS	10 (34%) {D ₁ ,D ₂ ,D ₄ ,D ₇ , D ₁₅ ,D ₁₈ ,D ₂₀ , D ₂₃ ,D ₂₄ ,D ₂₉ }	8 (28%) {D ₁ ,D ₂ ,D ₄ ,D ₅ , D ₁₈ -D ₂₀ ,D ₂₃ }	7 (24%) {D ₁ ,D ₂ ,D ₄ ,D ₅ , D ₁₅ ,D ₁₈ ,D ₂₀ }

Table 13: Summary comparison VTS x WTS about NTR values in the LIBSSH system.

	Time Budget 10%	Time Budget 50%	Time Budget 80%
Equivalent	6 (18%) {L ₃ ,L ₄ ,L ₉ ,L ₁₃ , L ₂₁ ,L ₂₆ }	3 (9%) {L ₃ ,L ₂₁ ,L ₂₇ }	3 (9%) {L ₃ ,L ₂₁ ,L ₃₂ }
VTS	5 (15%) {L ₁₀ ,L ₁₂ ,L ₁₅ , L ₁₇ ,L ₁₈ }	11 (32%) {L ₄ ,L ₉ ,L ₁₀ , L ₁₃ ,L ₁₅ ,L ₁₆ , L ₁₈ ,L ₁₉ ,L ₂₂ , L ₂₃ ,L ₂₉ }	16 (47%) {L ₄ ,L ₁₀ ,L ₁₁ , L ₁₃ ,L ₁₅ , L ₁₈ -L ₂₀ , L ₂₂ -L ₂₅ , L ₂₇ -L ₃₁ }
Max Reduction	98.53% {L ₈ }	93.88% {L ₂₄ }	95.23% {L ₂₄ }
WTS	23 (68%) {L ₁ ,L ₂ ,L ₅ -L ₈ , L ₁₁ ,L ₁₄ ,L ₁₆ , L ₁₇ ,L ₁₉ ,L ₂₀ , L ₂₂ -L ₂₅ , L ₂₇ -L ₃₄ }	20 (59%) {L ₁ ,L ₂ ,L ₅ -L ₈ , L ₁₁ ,L ₁₂ ,L ₁₄ , L ₁₇ ,L ₂₀ , L ₂₄ -L ₂₆ ,L ₂₈ , L ₃₀ -L ₃₄ }	15 (44%) {L ₁ ,L ₂ ,L ₅ -L ₉ , L ₁₁ ,L ₁₂ ,L ₁₄ , L ₁₆ ,L ₁₇ ,L ₂₆ , L ₃₃ ,L ₃₄ }
Max Reduction	99.90% {L ₈ }	99.90% {L ₈ }	99.90% {L ₈ }

Table 14: Summary comparison VTS x WTS about NTR values in the DUNE system.

	Time Budget 10%	Time Budget 50%	Time Budget 80%
Equivalent	1 (3%) {D ₁₄ }	2 (3%) {D ₁₈ ,D ₂₃ }	0 (0%)
VTS	16 (55%) {D ₃ ,D ₅ ,D ₆ , D ₈ -D ₁₂ ,D ₁₆ -D ₁₉ , D ₂₁ ,D ₂₅ ,D ₂₆ ,D ₂₈ }	19 (66%) {D ₃ ,D ₆ -D ₁₃ , D ₁₅ -D ₁₇ ,D ₁₉ , D ₂₁ ,D ₂₄ -D ₂₆ , D ₂₈ ,D ₂₉ }	20 (69%) {D ₃ ,D ₆ -D ₈ , D ₁₀ -D ₁₉ ,D ₂₁ , D ₂₃ -D ₂₆ ,D ₂₈ }
Max Reduction	99.91% {D ₃ }	99.93% {D ₃ }	99.92% {D ₃ }
WTS	12 (41%) {D ₁ ,D ₂ ,D ₄ ,D ₇ , D ₁₃ ,D ₁₅ ,D ₂₀ , D ₂₂ -D ₂₄ ,D ₂₇ ,D ₂₉ }	8 (28%) {D ₁ ,D ₂ ,D ₄ ,D ₅ , D ₁₄ ,D ₂₀ ,D ₂₂ ,D ₂₇ }	9 (31%) {D ₁ ,D ₂ ,D ₄ , D ₅ ,D ₉ ,D ₂₀ , D ₂₂ ,D ₂₇ ,D ₂₉ }
Max Reduction	100% {D ₁ ,D ₂ }	100% {D ₁ ,D ₂ }	100% {D ₁ ,D ₂ }

VTS is the best strategy in 169 cases for the time budgets 10%, 19 for 50%, and 20 for 80%. On the other hand, WTS is the best in 12 for

Table 15: The highest NTR differences between the strategies across the time budgets.

Variant	Time Budget 10%		Time Budget 50%		Time Budget 80%	
	VTS	WTS	VTS	WTS	VTS	WTS
LIBSSH						
L ₁	0.5152	0.9954	0.4618	0.9954	0.4495	0.9954
L ₅	0.5631	0.9540	0.6686	0.9540	0.7439	0.9540
L ₆	0.3935	0.9934	0.6004	0.9934	0.5052	0.9934
DUNE						
D ₂₀	0.4221	0.9997	0.6915	0.9997	0.7877	0.9997

Table 16: The worst performances for FRRMAB using VTS and WTS strategies regarding NAPFD metric.

Variant	Time Budget 10%		Time Budget 50%		Time Budget 80%	
	VTS	WTS	VTS	WTS	VTS	WTS
LIBSSH						
L ₁	0.6745	0.9706	0.5392	0.9706	0.4500	0.9706
L ₆	0.6235	0.9706	0.5647	0.9706	0.5294	0.9706
L ₈	0.5747	0.7656	0.5265	0.7706	0.5008	0.7674
L ₂₄	0.6310	0.6754	0.7450	0.8581	0.7609	0.8563
DUNE						
D ₁	0.7269	0.8145	0.5799	0.7954	0.5051	0.8590
D ₂	0.7776	0.9000	0.5803	0.9000	0.5307	0.9000
D ₂₀	0.4375	0.9955	0.7126	0.9955	0.7648	0.9955

10%, eight for 50%, and nine for 80%. Both strategies reach the maximum reduction in the maximum performance for the same variants across the time budgets. VTS for debian-11-gcc-9-17-python (D₃) and WTS for debian-11-gcc-9-17-downstream (L₁) and debian-11-gcc-9-17-downstream-dune-grid (L₂). Similar behavior is identified in the LIBSSH system, WTS presents high test time reduction in variants with only one commit.

Besides that, we identified that the highest differences of NTR values between the strategies (Table 15) are in variants with few historical test data. In this way, WTS provides higher performance when new variants are added in the system, and VTS the opposite.

Finding 13

Due its characteristics, WTS strategy allows mitigating some COLEMAN limitations: beginning without any knowledge in new variants and the existence of variants with few historical test data.

As we can observe with the facts aforementioned and in RQ1, some factors hamper a good prioritization but the main ones concern the historical test data and test case failure distribution. Such aspects align with the difficulty of an approach beginning without learning. To mitigate this, WTS should be used. As we can observe in the variants with the worst performance (Table 16), all of

them are using VTS strategy. When we consider the WTS strategy, performance increases considerably.

Finding 14

The prioritization is mainly impacted by the amount of historical test data and test case failure distribution.

In addition to the quantitative analysis, we performed a qualitative analysis to investigate the use of WTS as a prioritization strategy during the continuous integration of HCS. Tables 17 and 18 describe the history of reused and failed test cases along with the different builds. The second columns in the tables show the first build in which each variant was introduced. The third, fourth, and fifth columns present the number of test cases, the number of reused test cases, and new test cases applied during the continuous integration in that first build, respectively. In the last column we can observe the number of the reused test cases that have failed when used for testing other variants in previous builds and have been used used for testing that specific variant.

In the cases of variants in which test cases have never failed before the first build of the variants, both VTS and WTS have the same behavior. On the other hand, for a new variant whose test cases have been failed before its introduction, failed history can be an important source of information for the prioritization of test cases, then WTS can bring benefits for the CI process, mainly considering time constraints. For instance, in Table 17 we can see that the variants mingw64 and mingw64 of LIBSSH appear for the first time during build 45. All the 10 test cases of these variants were reused, since they were executed for testing other variants in previous builds. One of these test cases failed 16 times in previous builds of both variants. For DUNE (Table 18), the variant debian-11-gcc-9-17-downstream-dune-grid (last row) has failing test cases in 14 times before build 2169, in which this variant was introduced. In both cases, the test cases that failed in previous builds must be prioritized and executed before other test cases, since they are more likely to reveal faults.

For an overview of the testing history of LIBSSH and DUNE, Figure 11 presents the evolution of builds regarding the number of variants and test cases. The blue lines represent the number of variants, and as we can see, despite the systems have new variants over the time, the line remains constant, indicating volatility (inclusion and exclusion) of variants. Comparing the test cases available for the prioritization, the red lines indicate the amount of test cases available to be used when applying the strategy WTS. On the other hand, when using VTS, only a reduced number of test cases are available, which is indicated by the yellow lines. With VTS, the prioritization does not reuse the information from test cases of other variants already executed in the previous CI cycles.

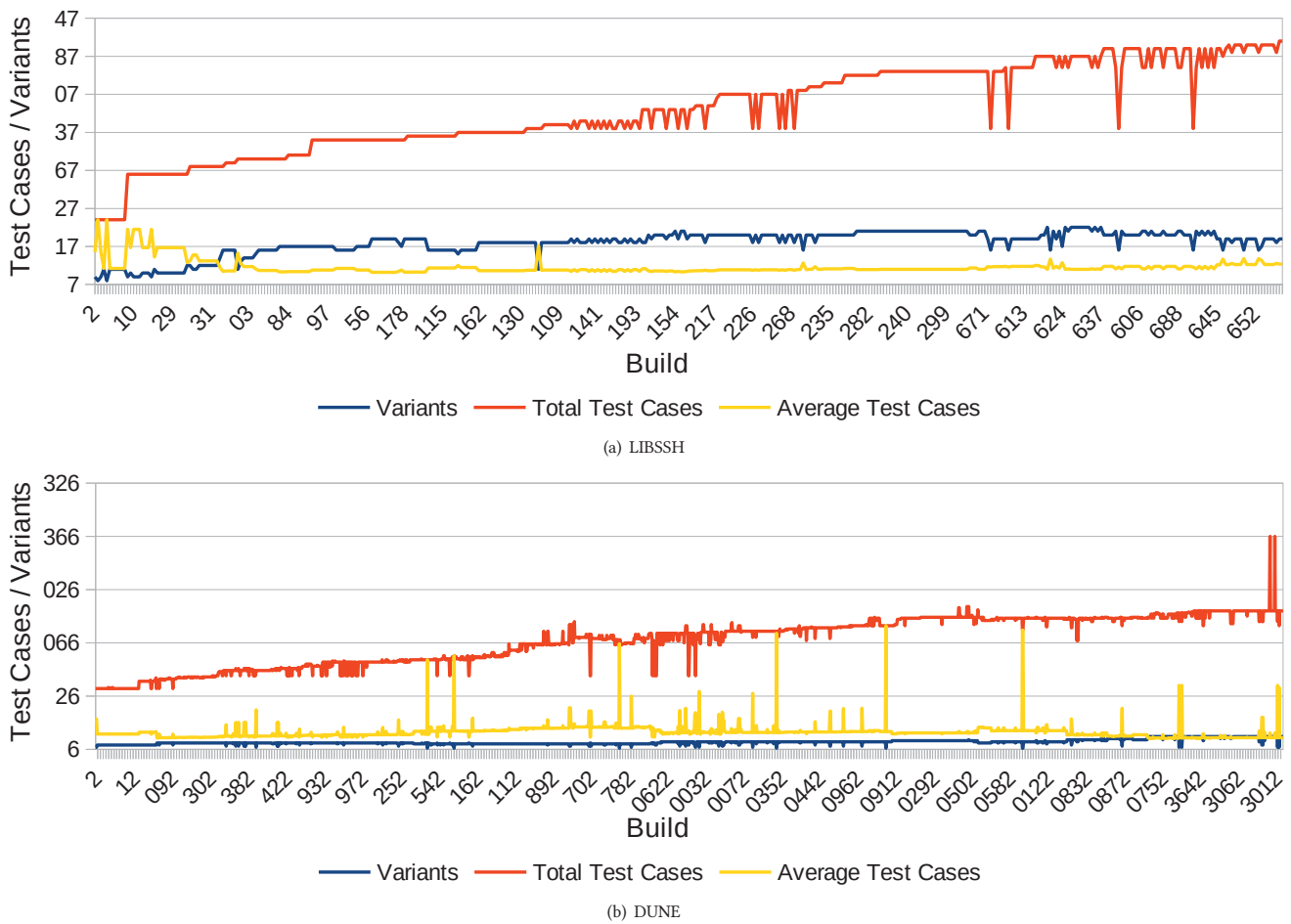


Figure 11: Number of variants and test cases per build

Finding 15

The history of test case failures and their reuse for testing new variants is a rich source of information to be considered during the test case prioritization in the context CI of HCSs. In this context, the problem of having volatility of variants along the CI cycles can be alleviated by adopting COLEMAN.

Answer for RQ5

Based on the quantitative and qualitative analysis of the results, we cannot point the best strategy. WTS provides better results with less restrictive situation (more time budget), and VTS the opposite, as well as taking into account NTR and PR values. The use of historical data information from the test cases reused across the variants benefits WTS. Because of this, WTS obtains better results than VTS in the first commits when there is not enough information to the learning, and when a new variant is added.

6.6 Implications and Limitations

This section discusses some implications of our findings to both practice and research. We also present some limitations that are in need of future investigation and threats to the validity of our results.

Table 17: Reused and Failed Reused Test Cases (TCs) along the build history of LIBSSH

Variant	First Build	Test Cases	Reused TCs	New TCs	Reused TCs Failed
CentOS7-openssl	2	17	0	17	0
Fedora-openssl	2	17	0	17	0
CentOS7-openssl_1_0_x-x86-64	3	17	17	0	2
Fedora-openssl_1_1_x-x86-64	4	17	17	0	3
Debian-openssl_1_0_x-aarch64	5	17	17	0	0
Fedora-libgrypt-x86-64	5	17	17	0	4
pages	18	29	17	12	0
address-sanitizer	22	29	29	0	19
undefined-sanitizer	22	29	29	0	53
centos7-openssl_1_0_x-x86-64	26	17	17	0	0
fedora-openssl_1_1_x-x86-64	26	29	29	0	0
fedora-undefined-sanitizer	26	29	29	0	0
tumbleweed-openssl_1_1_x-x86-64	33	29	29	0	0
tumbleweed-undefined-sanitizer	33	29	29	0	0
fedora-libgrypt-x86-64	44	31	29	2	0
fedora-mbedtls-x86-64	44	30	28	2	0
mingw64	45	10	10	0	16
mingw32	45	10	10	0	16
Debian_cross_mips-linux-gnu	64	19	18	1	0
fedora-openssl_1_1_x-x86-64-release	95	38	32	6	0
tumbleweed-openssl_1_1_x-x86-64-release	95	38	32	6	0
fedora-libgrypt-x86_64	131	40	38	2	0
centos7-openssl_1_0_x-x86_64	131	24	23	1	0
fedora-openssl_1_1_x-x86_64	131	40	38	2	0
fedora-mbedtls-x86_64	131	39	37	2	0
tumbleweed-openssl_1_1_x-x86_64-gcc	131	40	38	2	0
tumbleweed-openssl_1_1_x-x86_64-gcc7	131	40	38	2	0
tumbleweed-openssl_1_1_x-x86_64-clang	131	40	38	2	0
freebsd-x86_64	131	24	23	1	0
visualstudio-x86_64	188	16	11	5	0
visualstudio-x86	188	16	11	5	0
fedora-openssl_1_1_x-x86_64-minimal	197	40	39	1	0
fedora-openssl_1_1_x-x86_64-flips	258	55	46	9	0
ubuntu-openssl_1_1_x-x86_64	319	57	55	2	0

Table 18: Reused and Failed Reused Test Cases (TCs) along the build history of DUNE.

Variant	First Build	Test Cases	Reused TCs	New TCs	Reused TCs Failed
debian_9-gcc	1	56	0	56	0
debian_8-gcc	1	57	0	57	0
debian_9-clang	4	56	56	0	4
debian_8-clang	4	57	57	0	4
debian_8-backports-clang	114	64	57	7	0
ubuntu_16_04-gcc	114	64	57	7	0
ubuntu_16_04-clang	114	64	57	7	0
debian_10 gcc_c__17	467	82	63	19	0
debian_9 clang-3_8-14	1020	100	82	18	0
ubuntu_16_04 gcc-5-14	1020	100	82	18	0
ubuntu_18_04 clang-6-17	1020	100	82	18	0
debian_10 gcc-7-14-expensive	1020	109	82	27	0
debian_10 clang-6-libcpp-17	1020	104	82	22	0
debian_9 gcc-6-14	1022	104	104	0	0
ubuntu_16_04 clang-3_8-14	1027	98	97	1	0
debian_10 gcc-8-noassert-17	1040	100	100	0	0
debian_10 clang-7-libcpp-17	1123	101	100	1	0
debian_11 gcc-9-20	1465	106	100	6	0
debian_10 gcc-7-17	1606	117	106	11	0
debian_10 gcc-7-17-expensive	1632	124	121	3	0
debian-11-gcc-9-17-python	1648	110	110	0	3
ubuntu_20_04 clang-10-20	1787	113	113	0	4
ubuntu_18_04 clang-5-17	1787	113	113	0	0
debian_11 gcc-10-20	1850	114	113	1	0
ubuntu-20_04-gcc-9-17-python	1918	112	110	2	7
ubuntu_20_04 gcc-9-20	1935	117	117	0	0
ubuntu_20_04 gcc-10-20	1935	117	117	0	0
debian-11-gcc-9-17-downstream	2160	70	0	70	0
debian-11-gcc-9-17-downstream-dune-grid	2169	70	70	0	14

6.6.1 *Implications to Practice.* The implications below are related to the application of the proposed strategies.

1. *Approach to be used with VTS and WTS:* The evaluation of COLEMAN, RETECS, and Random using VTS and WTS strategies shows that the approaches have equivalent performance across the time budgets when a variant has few failures. Among the approaches, COLEMAN presents better performance, independently of the strategy used. In this way, COLEMAN can provide good prioritizations in the context of HCS in CI, dealing with the situation when there is a high test case volatility, and peaks or long periods without failures.

2. *VTS \times WTS - Performance regarding early fault detection and cost:* Analyzing the strategies, they present similar performance regarding NAPFD values under deterministic approach application. The WTS strategy can obtain prioritizations close or equivalent to those obtained by the VTS strategy. On the other hand, the strategies have opposite behaviors, in which the WTS strategy provides better NAPFD values with more time constraint (less time budget) and VTS the opposite. Furthermore, the strategies provide good results concerning the NAPFD metric. Thus, they contribute to reducing the test costs by decreasing the time spent in the CI Cycle. Among them, the prioritizations found by COLEMAN using the WTS strategy are reasonable solutions in 98% of the cases in both systems evaluated. Finally, both strategies are applicable in real scenarios with a low computational cost, spending around 0.042 seconds in the worst case to execute, and the time budget does not influence this time.

3. *VTS \times WTS - Performance regarding volatility of variants:* Regarding the characteristics of the strategies, we observed that the VTS strategy could provide good prioritizations. On the other hand, the historical test data across the variants is a rich source of information to be used when new variants are added. This is clear during the analysis of the WTS strategy. Such a strategy allows mitigating the problem in beginning without knowledge in new variants and those with few historical test data.

6.6.2 *Implications to Research.* Some implications are concerned to the limitations found, and serve as directions for future research.

1. *Performance improvement:* The worst performance reported by the approaches and strategies is the case where there are variants with a few CI Cycles and low failing ones. Another limitation is that COLEMAN and strategies do not consider the test time execution. Consequently, they have difficulty identifying test cases that fail but spend much time executing, which hamper good prioritization.

2. *Introduction of new strategies:* Regarding the strategies, the VTS strategy is directly connected to the approach characteristics. Thus, the limitations of this strategy are inherited from the approach used. On the other hand, the main characteristic of the WTS strategy is also a drawback. According to the prioritization order defined by such a strategy cannot be the best order for all variants. Some test cases that do not fail in some variants can appear before those failing ones. Thus, the WTS strategy should be used with COLEMAN to consider the knowledge from previous prioritizations. A possible research direction is to investigate ways to adapt or select a

strategy according to some characteristics of the systems, for instance, related to the failure distribution. To this end, meta-learning techniques could be used.

3. Variants Prioritization: In our work, we assume that there is a set of variants to be considered in the build and test phases of the CI cycle. But a research direction to be investigated is a strategy to establish an order of variants to be tested. Such an order should be used in cases where many variants are available and there are few resources for their execution. Some initiatives in the literature (see next section) for variant prioritization take into account factors such as similarity in terms of features and test cases to be reused, user preferences, testing criteria coverage, pairwise testing, and so on. However, the determination of these factors can imply costs. Then, an open question to be investigated in future work is using these factors to prioritize variants and their impact on the strategies VTS and WTS.

4. Scalability of the approaches: Another research question to be evaluated is the use of the proposed strategies with larger systems. It is expected a better performance of WTS with more variants, but more studies are needed to corroborate this finding. An open problem in the area is to propose strategies capable of dealing with huge configuration spaces with millions of variants. A research direction to be further investigated is using the strategies with techniques for variant selection and prioritization.

5. Benchmark construction: Our analysis revealed some interesting characteristics of the target systems and its variants that could be considered in the composition of a benchmark for future experiments. Although we used only two systems, they have an interesting number of variants that have different characteristics in relation to the test case volatility, variant volatility, historical test data, number of failures, number of failing cycles, test execution time. Furthermore, we observe a lack of studies and systems in context of HCS in CI. To build a benchmark for new studies is an important issue to be addressed by future works.

6.7 Threats to Validity

In this section we discuss the main threats to the validity of our results, and how we mitigate them. We use the taxonomy of Wohlin et al. [54].

Internal Validity evaluates the relationship between the treatment and the output. In our study, the parameter setting can be considered a threat. We used the same settings from literature [37, 38, 48], but it is possible that using a tuned configuration can lead to even better results. We did not perform the tuning in virtue of time constraints.

Our developed tool for mining the systems can be considered a threat. The interpretations that we made during the development can impact the results. To mitigate this, we adopted a pattern valid across the systems and variants to extract the tests. Besides that, the tool and the systems with the logs are available online to allow the replication.

Conclusion Validity is related to the ability to draw the correct conclusion from the study. An identified threat is the randomness. Ideally, it is recommended to execute the algorithms 1000

times [2]. However, this is not possible due to the computational effort required. For this, we considered a larger number of executions, namely 30 independent executions.

Another threat is related to the choice of the statistical tests used. The algorithms used are non-deterministic. We used tests commonly adopted in software engineering problems for this kind of algorithm to minimize this threat [6]. Finally, the analysis was made with a set of quality indicators. These results found may be different for other indicators. To minimize this threat, we used measures already used in the literature [37–39, 48].

External Validity corresponds to the ability to generalize the results beyond the experimental setting and whether our study's subjects are representative of real programs. In our study, we investigated two different open-source highly-configurable software systems. Thus, our results cannot be generalized. Unfortunately, we observe a lack of studies using systems in the context of HCS in CI. Besides that, it is a hard task finding systems with test results available to apply a data mining process, as well as logs with verbose information. In some occasions, the test result is available but there is no information about the test cases applied neither the test duration from each one. To mitigate this threat, we chose real-world systems, including several variants, real-world CI builds, test suites, and regression faults over a considerable period of time. Both systems have different characteristics concerning the number of commits, the number of failures, test case volatility, and variant volatility. In this way, our findings are valid within the investigated systems, and the study provides some evidence towards an initial validation of our strategies. To confirm our findings, new experiments should be performed; scalability for larger systems need to be evaluated. Furthermore, our study can be easily replicated using the raw data analyzed and disseminated by the Open Science Framework (OSF).

7 RELATED WORK

As we mentioned before, TCPCI for HCSs involves challenging particularities such as time constraints (test budgets), the volatility of test cases, and the volatility of variants. We find in the literature studies addressing TCP in the SPL/HCS context, but few of them address these CI challenges. In this section we first overviews work related to general SPL regression testing approaches, and after we focus on TCP approaches for SPL/HCS in the CI context. These last ones are the most related to ours. TCP approaches applied for CI, but not specific for SPL/HCS, were summarized in Section 3.

Pieces of work addressing regression testing and SPL engineering have been subject of systematic mapping and reviews [17, 46]. These studies explore three basic regression testing techniques, as classified by Yoo et al. [56]: test case minimization [51], test case selection [21, 22, 52, 55], and test case prioritization [1, 18, 19].

Approaches based on Test Case Minimization (TCM) usually remove redundant test cases, minimizing the test set according to some criterion. Some works use search-based algorithms to optimize a test suite considering time, revealing capability and coverage [51]. The use of these algorithms consume time and generally they are not suitable for the CI environment.

Test Case Selection (TCS) selects a subset of test cases, the most important ones. In the SPL context many works have as focus the

selection of products to be tested considering different goals such as: combinatorial testing, product similarity, coverage of variability and important features [12]. Approaches that have as focus test cases usually select a subset of existing test cases to be reused for testing a new product [22, 52, 55]. The problem with these pieces of work is that they consider differences between two products, and not the whole family. More recent regression testing studies are based on models and delta-oriented approaches [21], and code analysis [16]. The work of Lity et al. [21] captures commonality and variability of an evolving product line by means of differences between variants and versions of variants to select the test cases to be retested. The work of Jung et al. [16] proposes an automated code-based regression test selection method based on changes, and on commonality and variability to reuse of test cases. The advantage of this method is that it does not require specification or architectural model. But the dependency between test cases and source-code is not automatically determined.

The main disadvantage of test case minimization and selection techniques is that they do not consider the whole test set and may discard some important test cases. On the other hand, Test Case Prioritization (TCP) assume that all the test cases available may be executed. TCP attempts to re-order a test suite to identify an “ideal” order of test cases that maximizes specific goals, such as early fault detection.

The great majority of TCP existing work is devoted to the selection of the best product configurations to be tested, having the FM as starting point [17]. Some pieces of work consider TCP for SPL based on models with a delta-oriented approach. Lachmann et al. [19] showed an incremental delta-oriented approach for improving SPL integration testing efficiency by prioritizing test cases for product variants. Al-Hajjaji et al. [1] selected the most dissimilar product to the previously tested ones, in terms of deltas, to be tested next and studied the impact of adding delta modeling feature selection on product prioritization. Lachmann et al. [18] presented an approach for test case prioritization based on risk-based testing, which can automatically compute component failure impact and component failure probabilities for each product variant under test automatically. The problem of these TCP works is that they do not directly prioritize test cases. Such works do not consider the SPL evolution, that is, the different versions of variants in a regression testing scenario.

Hajri et al. [15] present an automated test case classification and prioritization approach based on our use case-driven modeling and configuration techniques. The approach adopts the incremental testing. For new products in the SPL, it automatically classifies and prioritizes test cases of previous products, and provides guidance in modifying existing system test cases to cover new use case scenarios that have not been tested in the product line before. Test cases are prioritized based on a logistic regression model considering multiple risk factors such as fault-proneness of requirements and requirements volatility. This approach can be classified as to learn-to-rank approach, but it assumes a use case-driven development and is model-dependent.

The pieces of works mentioned above do not focus CI particularities and constraints, but the regression testing techniques are not exclusive [12], and some of the proposed approaches can be

applied in a previous step, in a complementary way to establish a test set for a build to be prioritized by our strategies.

The work of Pett et al. [34] introduces a metric to measure the stability of sampling algorithms in the context of SPL regression testing. Considering that SPL products should be sampled for the regression testing during each CI cycle, it is desired to have similar products from one cycle to another. Despite focusing on CI and SPL this work does not focus on test case prioritization. Also, there is no discussion regarding the variant volatility and test case volatility across the CI cycles in this study.

The work of Marijan et al. [27] introduces ROCKET, an approach that sets a weight for each test case based on the distance of the failure status from its current execution and its execution time. But the approach does not consider prioritization feedback nor the total history of failures. Test cases with an execution time greater than a limit are penalized, and it is possible they are never executed. An extension is proposed by the authors in [24] to consider other perspectives regarding fault detection, business, performance and technical aspects. Such an extension needs additional information related to coverage and features. An approach and a tool called TITAN is proposed by the authors in [29]. The tool implements test prioritization and minimization techniques, and provides test traceability and visualization. The idea is to obtain a high fault detection rate and low test execution. The approach considers that the test cases have tags for HCS features. However, in most open-source HCS systems these macros do not exist. This hampers the applicability of the approach for general scenarios.

Another work of Marijan et al. [26] uses the coverage matrix of test cases and the fault detection history to identify redundant test cases that are not likely to detect faults. Their method minimizes a test suite by excluding redundant test cases. It is a learning algorithm that reduces test redundancy. The algorithm minimizes the test execution time by avoiding unnecessary test executions using coverage metrics and a fault-detection history. Again, additional information, such as feature coverage, is necessary. As firstly a minimization step is conducted, the prioritized set may not contain all available test cases

In summary, most TCP approaches [1, 17–19] have as focus the prioritization of products to be tested; they do not directly prioritize test cases. Incremental and delta-oriented approaches can be applied to select test cases (or reuse) when new products are added. Despite these approaches do not address the CI context, some TCM and TCS approaches can be used in a combined way with our approach, as a previous step for establish the set test case or products for a build. On the other hand, approaches that have been applied in the CI context are model dependent [15] or do not properly deal with the EvE problem [25, 27, 29]. This problem regards to the fact that as only a sub-set of the prioritized test cases can be executed regarding its order, some test cases can never be executed given the test budget. To deal with this, most approaches use, besides the failure-history, other measures that rely on code instrumentation or require additional information, such as to calculate code or feature coverage. This can be time-consuming and to maintain the information updated can be difficult.

Ranking-to-learning approaches like COLEMAN and RETECS deal properly with the EvE problem and test case volatility but they do not consider the HCS context. Our work has as main contribution

to allow the use of such approaches, initially designed only for TCP in CI, to TCP in CI of HCS, by proposing two strategies: VTS and WTS. These strategies take into account particularities, such as the variant volatility. In addition to this we provide results evaluating both strategies, and comparing their performance using COLEMAN and RETECS, and a deterministic approach.

8 CONCLUDING REMARKS

This work presents and evaluates two strategies for the application of a TCP learning-based approach, COLEMAN, in Continuous Integration of HCSs: the Variant Test Set Strategy (VTS) that relies on the test set specific for each variant, and the Whole Test Set Strategy (WTS) that prioritizes the test set composed by the union of the test cases of all variants. COLEMAN uses a MAB policy and a reward function to learn from the failure-history of test cases, addressing, in this way, the volatility problem, regarding test cases and variants that can be added or removed along the CI cycles.

We evaluated the strategies in two real-world systems, using the FRRMAB policy and TimeRank reward function and considering three time budgets, namely 10%, 50%, and 80%. Our evaluation was guided by five RQs. RQ1 and RQ2 evaluated the use of COLEMAN in comparison with respectively, a random prioritization and the learning-based approach RETECS. The results show that COLEMAN overcomes both approaches in terms of NAPFD, independently of the budget and strategy adopted. COLEMAN presents the best results or statistically equivalent to the random approach in 96% of the cases for the LIBSSH system and 100% for the DUNE system. Compared with RETECS, COLEMAN obtained the best results or statistically equivalent ones in 98% of the cases for both systems.

The results of RQ3 and RQ4 show the strategies used with COLEMAN are very cost-effective. In comparison with the deterministic approach, COLEMAN presents reasonable solutions in the LIBSSH system in 93% and 98% of the cases using respectively the VTS and WTS strategies, that is, its solutions are very near or near to the optimal ones, against 90% and 82% produced by the random approach, and 78% and 72% produced by RETECS. Regarding the DUNE system, COLEMAN using the VTS strategy produces 92% of reasonable solutions, and 98% using the WTS strategy. On the other hand, random prioritization produces only 63% and 64%, respectively, and RETECS produces 45%. Furthermore, the strategies spend, in the worst case, just 0.0396 seconds to execute for LIBSSH and 0.0418 seconds for DUNE, what shows their applicability in the CI context. We then can conclude that the strategies

Results of RQ5 comparing both strategies show they present similar performance considering the indicators. But WTS performs better in a more restrictive scenario, i.e., time budget of 10%, and with the VTS strategy occurs the opposite. Besides that, WTS better mitigates the problem of beginning without knowledge. Consequently, this strategy is adequate when there is a new variant to be tested.

As future work we intend to investigate some research directions discussed in Section 6.6, as well as to address some limitations. We should conducted new evaluations applying the strategies for other HCSs from different domains and with different number of variants. We also intend to apply COLEMAN with other policies, as well as other features such as test coverage, testers' preference, test case

duration and, to consider in how many variants a test case fails. Specific metrics to the variants could be explored.

ACKNOWLEDGMENTS

This research was partially funded by the Brazilian research agencies: CNPq (grants 408356/2018-9 and 305968/2018-1), Fundação Araucária – FAPPR (grant no. 51435), FAPERJ PDR-10 program (grant no. 202073/2020), and CAPES.

CONFLICT OF INTEREST

The authors declare that they have no conflict of interest.

REFERENCES

- [1] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-oriented product prioritization for similarity-based product-line testing. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 34–40.
- [2] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *33rd International Conference on Software Engineering (Waikiki, Honolulu, HI, USA) (ICSE'11)*. ACM, New York, NY, USA, 1–10. <https://doi.org/10.1145/1985793.1985795>
- [3] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *42nd International Conference on Software Engineering (Seoul, Republic of Korea) (ICSE'20)*. ACM, New York, NY, USA.
- [4] R. Capilla, J. Bosch, and K.C. Kang. 2013. *Systems and Software Variability Management: Concepts, Tools and Experiences*. Springer.
- [5] Y. Cho, J. Kim, and E. Lee. 2016. History-Based Test Case Prioritization for Failure Information. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 385–388. <https://doi.org/10.1109/APSEC.2016.066>
- [6] Thelma Elita Colanzi, Wesley Klewerton Guez Assunção, Paulo Roberto Farah, Silvia Regina Vergilio, and Giovani Guizzo. 2019. A Review of Ten Years of the Symposium on Search-Based Software Engineering. In *Symposium on Search-Based Software Engineering*. Springer, Cham, 42–57.
- [7] Judith E Dayhoff. 1990. *Neural network architectures: an introduction*. Van Nostrand Reinhold Co.
- [8] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. 2008. An Empirical Study of the Effect of Time Constraints on the Cost-Benefits of Regression Testing. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Atlanta)*. ACM, New York, NY, USA, 71–82.
- [9] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
- [10] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (Hong Kong)*. ACM, New York, NY, USA, 235–245. <https://doi.org/10.1145/2635868.2635910>
- [11] Emelie Engström. 2010. Regression Test Selection and Product Line System Testing. In *3rd International Conference on Software Testing, Verification and Validation*. IEEE, 512–515.
- [12] Alireza Ensan, Ebrahim Bagheri, Mohse Asadi, Dragan Gasevic, and Yevgen Biletskiy. 2011. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *8th International Conference on Information Technology: New Generations*. IEEE, 291–298. <https://doi.org/10.1109/ITNG.2011.58>
- [13] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. 2016. Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing. In *8th International Symposium on Search Based Software Engineering*. Springer, Cham, 49–63.
- [14] Alireza Haghighatkah, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98. <https://doi.org/10.1016/j.jss.2018.08.061>
- [15] Ines Hajri, Arda Goknil, Fabrizio Pastore, and Lionel C Briand. 2020. Automating system test case classification and prioritization for use case-driven testing in product lines. *Empirical Software Engineering* 25, 5 (2020), 3711–3769.
- [16] Pilsu Jung, Sungwon Kang, and Jihyun Lee. 2019. Automated code-based test selection for software product line regression testing. *Journal of Systems and Software* 158 (2019), 110419. <https://doi.org/10.1016/j.jss.2019.110419>
- [17] Satendra Kumar and Rajkumar. 2016. Test case prioritization techniques for software product line: A survey. In *International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 884–889.

- [18] Remo Lachmann, Simon Beddig, Sascha Lity, Sandro Schulze, and Ina Schaefer. 2017. Risk-based integration testing of software product lines. In *11th International Workshop on Variability Modelling of Software-intensive Systems*. 52–59.
- [19] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. 2015. Delta-oriented test case prioritization for integration testing of software product lines. In *19th International Conference on Software Product Line*. 81–90.
- [20] K. Li, A. Fialho, S. Kwong, and Q. Zhang. 2014. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on* 18, 1 (2014), 114–130.
- [21] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. 2019. Retest test selection for product-line regression testing of variants and versions of variants. *Journal of Systems and Software* 147 (2019), 46–63.
- [22] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. 2012. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *Tests and Proofs*. Springer, Berlin, Heidelberg, 67–82.
- [23] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* 18 (1947), 50–60.
- [24] Dusica Marijan. 2015. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (Washington, DC, USA) (QRS)*. IEEE Computer Society, 157–162. <https://doi.org/10.1109/QRS.2015.31>
- [25] Dusica Marijan, Arnaud Gotlieb, and Marius Liaen. 2019. A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience* 49, 2 (2019), 192–213.
- [26] Dusica Marijan, Arnaud Gotlieb, and Marius Liaen. 2019. A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience* 49, 2 (2019), 192–213. <https://doi.org/10.1002/spe.2661>
- [27] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *IEEE International Conference on Software Maintenance (ICMS)*. IEEE, 540–543.
- [28] Dusica Marijan and Marius Liaen. 2017. Test Prioritization with Optimally Balanced Configuration Coverage. In *IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 100–103.
- [29] Dusica Marijan, Marius Liaen, Arnaud Gotlieb, Sagar Sen, and Carlo Ieva. 2017. TITAN: Test Suite Optimization for Highly Configurable Software. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 524–531. <https://doi.org/10.1109/ICST.2017.60>
- [30] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *38th International Conference on Software Engineering (Austin, Texas) (ICSE '16)*. ACM, New York, NY, USA, 643–54. <https://doi.org/10.1145/2884781.2884793>
- [31] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Balduino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469.
- [32] Mukelabai Mukelabai, Damir Nesiundefin, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France)*. ACM, New York, USA, 155–166. <https://doi.org/10.1145/3238147.3238201>
- [33] Raiza Oliveira, Bruno Cafeo, and Andre Hora. 2019. On the Evolution of Feature Dependencies: An Exploratory Study of Preprocessor-Based Systems. In *13th International Workshop on Variability Modelling of Software-Intensive Systems (Leuven, Belgium)*. ACM, New York, NY, USA, 1–9. <https://doi.org/10.1145/3302333.3302342>
- [34] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *15th international Conference on Variability Modelling of Software-Intensive Systems (Krems, Austria) (VaMoS'21)*. Association for Computing Machinery, Article 18, 9 pages. <https://doi.org/10.1145/3442391.3442410>
- [35] Jackson A. Prado Lima, Willian DF Mendonça, Silvia R Vergilio, and Wesley KG Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*. 1–11.
- [36] Jackson A. Prado Lima, Willian D. F. Mendonça, Wesley K. G. Assunção, and Silvia R. Vergilio. 2021. Supplementary Material - Cost-effective learning-based strategies for test case prioritization in Continuous Integration of Highly-Configurable Software. https://osf.io/z3r2e/?view_only=db9ab0ed2b8e4289b22d4ad0c83c13c1
- [37] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments. *IEEE Transactions on Software Engineering* (2020), 12. <https://doi.org/10.1109/TS.E.2020.2992428>
- [38] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. Multi-Armed Bandit Test Case Prioritization in Continuous Integration Environments: A Trade-off Analysis. In *5th Brazilian Symposium on Systematic and Automated Software Testing (SAST'20)*. ACM. <https://doi.org/10.1145/3425174.3425210>
- [39] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020), 106268. <https://doi.org/10.1016/j.infsof.2020.106268>
- [40] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. 2007. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *IEEE International Conference on Software Maintenance*. IEEE, 255–264. <https://doi.org/10.1109/ICSM.2007.4362638>
- [41] Hanna Remmel, Barbara Paech, Peter Bastian, and Christian Engwer. 2011. System testing a scientific framework using a regression-test environment. *Computing in Science & Engineering* 14, 2 (2011), 38–45.
- [42] Hanna Remmel, Barbara Paech, Christian Engwer, and Peter Bastian. 2013. Design and rationale of a quality assurance process for a scientific framework. In *2013 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*. IEEE, 58–67.
- [43] Herbert Robbins. 1985. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*. Springer, 169–177.
- [44] Gregg Rothermel. 2018. Improving Regression Testing in Continuous Integration Development Environments (Keynote). In *9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (Lake Buena Vista, FL, USA) (A-TEST 2018)*. ACM, New York, NY, USA, 1. <https://doi.org/10.1145/3278186.3281454>
- [45] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *IEEE International Conference on Software Maintenance (Washington, DC, USA) (ICSM '99)*. IEEE Computer Society, 179.
- [46] Per Runeson and Emelie Engström. 2012. Chapter 7 - Regression Testing in Software Product Line Engineering. In *Advances in Computers*, Ali Hurson and Atif Memon (Eds.). Vol. 86. Elsevier, 223–263. <https://doi.org/10.1016/B978-0-12-396535-6.00007-7>
- [47] M. Shahin, M. Ali Babar, and L. Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943.
- [48] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *26th ACM SIGSOFT International Symposium on Software Testing and Analysis (Santa Barbara, CA, USA) (ISSTA 2017)*. ACM, New York, NY, USA, 12–22. <https://doi.org/10.1145/3092703.3092709>
- [49] Andras Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (Jan. 2000), 101–132.
- [50] Alexander von Rhein, Alexander Grebhorn, Sven Apel, Norbert Siegmund, Dirk Bayer, and Thorsten Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *37th International Conference on Software Engineering - Volume 1 (Florence, Italy) (ICSE 2015)*. IEEE, New York, USA, 178–188.
- [51] Shuai Wang, Shaikat Ali, and Arnaud Gotlieb. 2015. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software* 103 (2015), 370–391. <https://doi.org/10.1016/j.jss.2014.08.024>
- [52] Shuai Wang, Shaikat Ali, Arnaud Gotlieb, and Marius Liaen. 2016. A systematic test case selection methodology for product lines: results and insights from an industrial case study. *Empirical Software Engineering* 21 (2016), 1586–1622.
- [53] T. Wang and T. Yu. 2018. A Study of Regression Test Selection in Continuous Integration Environments. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 135–143.
- [54] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, USA. <https://doi.org/10.1007/978-1-4615-4625-2>
- [55] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. 2013. Continuous Test Suite Augmentation in Software Product Lines. In *17th International Software Product Line Conference (Tokyo, Japan) (SPLC '13)*. Association for Computing Machinery, New York, NY, USA, 52–61. <https://doi.org/10.1145/2491627.2491650>
- [56] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software: Testing, Verification, and Reliability* 22, 2 (March 2012), 67–120. <https://doi.org/10.1002/stvr.430>
- [57] Zhi Quan Zhou, Chen Liu, Tsong Yueh Chen, T. H. Tse, and Willy Susilo. 2021. Beating Random Test Case Prioritization. *IEEE Transactions on Reliability* 70, 2 (2021), 654–675. <https://doi.org/10.1109/TR.2020.2979815>