Universitá degli Studi di Salerno

Dipartimento di Studi e Ricerche Aziendali

Dottorato di Ricerca in Scienze e Tecnologogia dell'Informazione,
dei Sistemi Complessi e dell'Ambiente

XII Ciclo

Tesi di dottorato
in
Software Engineering

Search-Based Software Maintenance and Testing

Annibale Panichella

PhD Program Coordinator
Prof. Roberto Scarpa

Supervisors
Prof. Andrea De Lucia

Aprile 2014

Acknowledges

Summarising more than three years of work and experience is not a trivial task and finding the right words to express acknowledgements is quite complex. I hope that they do not lookalike a set of ritual sentences, while I really feel and mean every single word.

Foremost I am deeply grateful to my advisors, Prof. Andrea De Lucia and Dr. Rocco Oliveto, for the support of my Ph.D study, for their patience, motivation and enthusiasm. It has been a honour to be advised by Andrea who tirelessly provided me encouragement, advise and meticulous comments necessary in all the time of research and writing of this thesis. Special thanks to Rocco who has been my advisor and mentor through many years of university study, from the Bachelor to my PhD study. I have also to thank him for believing in me and for the encouragement to continue my study, to start and complete my PhD. Other than of being excellent supervisors, Andrea and Rocco are very good friends to me and thanks to their support I feel myself improved from both professional and personal point of views. Special thanks to my examiners, Prof. Giuliana Vitiello (University of Salerno), Prof. Gerardo Canfora (University of Sannio) and Prof. Raffaele Savino (University of Naples "Federico II") for managing to read this dissertation and for their helpful suggestions.

I owe my deepest gratitude to Prof. Massimiliano Di Penta of the University of Sannio for the support during my research works. I am truly thankful for his steadfast integrity, enthusiasm and dedication, which concurred to my personal and academic development. I have to thank him for the discussions, the interactions and the extensive comments that contributed to the quality and the content of this thesis. My sincere thanks also goes to Prof. Denys Poshyvanik of The College of William and Mary for offering me the summer internship opportunities in his research group and for supporting me on diverse exciting works. I would like to thank Prof. Paolo Tonella of the Fondazione Bruno Kessler for giving me constructive comments, warm encouragement and enormous contribution to my works on Software Testing. I would like to offer my special thanks to Dr. Giovanni Campobianco from University of Molise for showing me the beautiful of math and its applications to many real-world problems. I have greatly benefited from his enthusiasm and generous moral support.

I had the pleasure to work with several students who somehow contributed to the work

presented in this dissertation. Bogdan Dit worked with me during the design and development of search-based solutions for program comprehension presented in Chapter 3. Fitsum M. Kifetew implemented some components of the Evosuite prototype presented in Chapter 7 in the context of evolutionary test data generations.

Last but not the least, I would like to thank my family: my parents, Salvatore Panichella (who unfortunately has passed away during my first year as PhD student) and Anna D'Annessa. Their love is my inspiration and is my driving force. I would like to thank my twin, Sebastiano, brother and best friend against the many adversities of the life. Special thanks for the support, admirable enthusiasms, honesty and help he always provide me. He was my source of strength when I was weak and vulnerable. Thanks to my sister, Lucia, for her unique sense of humour, love and incredible strength. I am glad for the jokes, the walks and the innumerable discussions we had all over together.

I am also grateful to all my friends for being my second family and for their continuous moral support.

List of related publications

List of journal publications made by the candidate during the three years PhD work.

[**1**] Andre De Lucia, Massimiliano Di Penta, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **Labeling source code with information retrieval methods: an empirical study**. Empirical Software Engineering. In press.

[**2**] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **Applying a smoothing filter to improve IR-based traceability recovery processes: An empirical investigation**. Information and Software Technology 2012.

[**3**] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **Improving IR-based traceability recovery via noun-based indexing of software artifacts**. Journal of Software: Evolution and Process 2013.

List of conference publications.

[**4**] *Annibale Panichella*, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, Andrea De Lucia. **How to Effectively Use Topic Models for Software Engineering Tasks? An Approach based on Genetic Algorithms**. In proceedings of the 35th IEEE/ACM International Conference on Software Engineering (ICSE). San Francisco, CA, USA, 2013.

[**5**] Fitsum M. Kifetew, *Annibale Panichella*, Andrea and De Lucia, Rocco Oliveto, Paolo Tonella. **Orthogonal Exploration of the Search Space in Evolutionary Test Case Generation**. In proceedings of the International Symposium on Software testing and analysis (ISSTA). Lugano, Switzerland, 2013.

[**6**] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **Multi-Objective Cross-Project Defect Prediction**. In proceedings of the International Conference on Software Testing, Verification and Validation (ICST). Luxemburg, 2013.

[**7**] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, *Annibale Panichella*. **Estimating the evolution direction of populations to improve genetic algorithms**. In proceedings of the Genetic and Evolutionary Computation Conference (GECCO). Philadelphia, USA, 2012.

[**8**] Gabriele Bavota, Luigi and Colangelo, Andrea De Lucia, Sabato Fusco, Rocco Oliveto, *Annibale Panichella*. **TraceME: Traceability Management in Eclipse**. In proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM). Trento, Italy, 2012.

[**9**] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **Improving IR-based Traceability Recovery Using Smoothing Filters**. In proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC), Kingston, ON, Canada, 2011.

[**10**] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **Using IR methods for labeling source code artifacts: Is it worthwhile?** In proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC). Passau, Germany, 2012.

[**11**] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella. **On the role of diversity measures for multi-objective test case selection**. In proceedings of the 7th International Workshop on Automation of Software Test (AST). Zurich, Switzerland, 2012.

[**12**] *Annibale Panichella*, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, Andrea De Lucia. **When and How Using Structural Information to Improve IR-Based Traceability Recovery**. In proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR). Genova, Italy, 2013.

[**13**] *Annibale Panichella*, Rocco Oliveto, and Andrea De Lucia. **Cross-Project Defect Prediction Models: L'Union Fait la Force**. In proceedings of the IEEE CSMR-WCRE Software Evolution Week. Antwerp, Belgium, 2014.

[**14**] Bogdan Dit, *Annibale Panichella*, Evan Moritz, Rocco Oliveto, Massimilano Di Penta, Denys Poshyvanyk, Andrea De Lucia. **Configuring topic models for software engineering tasks in TraceLab**. In proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE). Los Alamitos, CA, USA, 2013.

[**15**] Gabriele Bavota, Andrea De Lucia, Rocco Oliveto, *Annibale Panichella*, Fabio Ricci, Genoveffa Tortora. **The role of artefact corpus in LSI-based traceability recovery**. In proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE). Los Alamitos, CA, USA, 2013.

[16] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **Traceability recovery using numerical analysis**. In Proceedings of the 16th Working Conference on Reverse Engineering (WCRE), 2009.

[17] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, *Annibale Panichella*, Sebastiano Panichella. **On the Role of the Nouns in IR-based Traceability Link Recovery**.In Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC), 2009.

The following papers were made by the candidate during the three years PhD work and are currently under review process in journals:

[18] *Annibale Panichella*, Rocco Oliveto, Massimilano Di Penta, Andrea De Lucia. **Improving Multi-Objective Search Based Test Suite Optimization through Diversity Injection**. Submitted to Transactions on Software Engineering (TSE).

[19] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, Sebastiano Panichella. **Defect Prediction as a Multi-Objective Optimization Problem**. Actually considered with Major revision in Software Testing, Verification and Reliability (STVR).

In the following, for each chapter it is specified the used materials (or part) of the publications presented in this thesis:

Chapter 3 Publication [1, 2, 3, 14, 4, 9, 10, 12, 15, 8]

Chapter 4 Publication [6, 19]

Chapter 6 Publication [7]

Chapter 7 Publication [5, 7]

Chapter 8 Publication [5, 7, 11, 18]

Contents

1 Introduction **1**

1.1 Research problem and motivations 2
1.2 Major Contributions 4
1.2.1 IR-based program comprehension using search algorithm 4
1.2.2 Multi-objective defect prediction 5
1.2.3 Improving search-based software testing through diversity injection . . 6
1.2.4 Improving test suite optimization through diversity injection 7
1.3 Thesis organization 7

I Software Maintenance **9**

2 Background: Genetic Algorithms **11**

2.1 Introduction and Motivations 12
2.2 Search-based approaches to Software Maintenance 13
2.3 Optimization problems 15
2.4 Evolutionary Computation 17
2.5 Single-objective Genetic Algorithms 19
2.5.1 Solution encoding 20
2.5.2 Selection operators 21
2.5.3 Crossover operators 22
2.5.4 Mutation operators 24
2.5.5 Elitism 25
2.5.6 Termination 25
2.6 Multi-objective Genetic Algorithms 26
2.6.1 NSGA-II 27

3 Finding optimal IR process using GAs **29**

3.1 Introduction and Motivation 31
3.2 Background and Related work 33
3.2.1 IR methods 35

3.2.2 IR-based traceability recovery . . . 38
3.2.3 IR-based feature location . . . 39
3.2.4 IR-based bug report duplication . . . 40
3.2.5 IR-based source code labeling . . . 41
3.2.6 Choosing the right IR process . . . 42
3.3 Relationship between Clustering Quality and Performances of IR-based techniques . . . 43
3.4 Finding a (Near) Optimal IR process . . . 47
3.5 Finding a (Near) Optimal LDA Configuration . . . 49
3.6 Empirical Evaluation of LSI-GA . . . 51
3.6.1 Research Questions . . . 51
3.6.2 Scenario I: Traceability Recovery . . . 52
3.6.3 Scenario II: Feature Location . . . 56
3.6.4 Scenario III: Duplicated Bug Report . . . 60
3.7 Empirical Evaluation of LDA-GA . . . 63
3.7.1 Research Questions . . . 63
3.7.2 Scenario I: Traceability Recovery . . . 64
3.7.3 Scenario II: Feature Location . . . 68
3.7.4 Scenario II: Source Code Labeling . . . 72
3.8 Threats to Validity . . . 74
3.9 Conclusion and Future Work . . . 76

4 Multi-Objective Cross-project Defect Prediction 77
4.1 Introduction and Motivation . . . 79
4.2 Background and Related work . . . 80
4.2.1 Metrics . . . 81
4.2.2 Machine Learning Techniques . . . 83
4.2.3 Cross-project defect prediction . . . 84
4.3 MODEP: Multi-Objective Defect Predictor . . . 86
4.3.1 Data Preprocessing . . . 86
4.3.2 Multi-Objective Defect Prediction Models . . . 87
4.3.3 Training the Multi-Objective Predictors using Genetic Algorithms . . . 92
4.4 Design of the Empirical Study . . . 93
4.4.1 Definition and Context . . . 93
4.4.2 Research Questions . . . 94
4.4.3 Variable Selection . . . 95
4.4.4 Analysis Method . . . 96
4.4.5 Implementation and Settings of the Genetic Algorithm . . . 98
4.5 Study Results . . . 99
4.5.1 RQ_1: *How does MODEP perform compared to single-objective prediction?* 99
4.5.2 RQ_2: *How does MODEP perform compared to the local prediction approach?* . . . 107

4.5.3 Benefits of MODEP as Compared to Single-Objective Defect Predictors 109
4.6 Threats to Validity 111
4.7 Conclusion and Future Work 112

II Software Testing 115

5 Software Testing: Background and Related Work 117
5.1 Introduction 118
5.2 Test data generation 119
5.2.1 Structural testing and basic concepts 120
5.2.2 Coverage criteria and flag problem 122
5.2.3 Solution Encoding for Test Data Generation 124
5.2.4 Fitness functions for test data generation 125
5.3 Test suite optimization 128
5.3.1 Problems definition 129
5.3.2 Test Suite Optimization with Traditional Approaches 131
5.3.3 Search-based Test Suite Optimization 132
5.4 Diversity and population drift 133
5.5 Improving evolutionary testing by diversity injection 136

6 Improving genetic algorithms through diversity injection 139
6.1 Introduction 140
6.2 Injecting Diversity using SVD 141
6.2.1 Estimating the evolution directions using SVD 141
6.2.2 Generating orthogonal individuals 144
6.2.3 Integrating SVD with genetic algorithms 146
6.2.4 SVD complexity 148
6.3 Empirical study on single-objective GAs 148
6.3.1 Data Collection and Analysis 149
6.3.2 Empirical Results 150
6.4 Empirical study on MOGAs 153
6.4.1 Data Collection and Analysis 154
6.4.2 GA Parameter Settings 155
6.4.3 Empirical Results 156
6.5 Conclusion 161

7 Test Data Generation 163
7.1 Introduction 164
7.2 Diversifying the Exploration 165
7.2.1 Basic SVD-GA 167
7.2.2 History Aware SVD-GA 167

7.2.3 Reactive GA 168
7.2.4 Reactive SVD-GA 168
7.3 Implementation 168
7.4 Empirical Evaluation 172
7.4.1 Subjects 172
7.4.2 Research Questions 173
7.4.3 Metrics and Data Analysis 174
7.4.4 GA Parameter Settings 175
7.5 Analysis of the Results 176
7.5.1 **RQ**$_1$: *Does orthogonal and/or reactive exploration improve the effectiveness of evolutionary test case generation?* 176
7.5.2 **RQ**$_2$: *Does orthogonal and/or reactive exploration improve the efficiency of evolutionary test case generation?* 179
7.6 Threats to validity 181
7.7 Conclusion and future work 183

8 Multi-Objective Test Suite Optimization 185
8.1 Introduction and Motivation 186
8.2 Injecting Diversity in Multi-Objective Test Suite Optimization: DIV-GA . . . 187
8.2.1 The DIV-GA algorithm 190
8.3 Empirical Evaluation 191
8.3.1 Research Questions 192
8.3.2 Test Selection Objectives 193
8.3.3 Experimented Algorithms 195
8.3.4 Evaluation Metrics 197
8.4 Empirical Results 200
8.4.1 **RQ**$_1$: *To what extent does DIV-GA produce near optimal solutions, compared to alternative test case selection techniques?* 201
8.4.2 **RQ**$_2$: *What is the cost-effectiveness of DIV-GA, compared to alternative test case selection techniques?* 208
8.5 Threats to Validity 211
8.6 Conclusion and Future Work 212

9 Conclusion and feature work 215
9.1 Summary of Contributions 216
9.2 Future Work 218

A SVD-GA: Performances of the Experimented Algorithms 221
A.1 Empirical study on single-objective GAs 221
A.2 Empirical study on MOGAs 221
A.2.1 Graphs of Non-Dominated Solutions obtained by SVD-NSGA-II and NSGA-II 222

B Binary Representation of Testing Criteria **231**

C Multi-Objective Test Suite Optimization **235**

C.1 **RQ_1**: To what extent does DIV-GA produce near optimal solutions, compared to alternative test case selection techniques? 235

C.2 **RQ_2**: What is the cost-effectiveness of DIV-GA, compared to alternative test case selection techniques? 242

References **248**

List of Figures

2.1 Example of roulette wheel selection 21

3.1 Outline of a generic IR Process to solve SE problems. 34
3.2 LDA process 37
3.3 Examples of different textual clustering 45
3.4 Example of the Silhouette coefficients. 47
3.5 LSI-GA chromosome representation. 48
3.6 Traceability recovery: precision/recall graphs. 55
3.7 Box plots of the effectiveness measure for feature location on jEdit and JabRef. 58
3.8 Recall Rate graphs for Eclipse, with suggested list size raging between 1 and 25. 62
3.9 Variability of performance achieved by LDA configurations for traceability link recovery 66
3.10 Traceability recovery: precision/recall graphs. 67
3.11 Variability of performance achieved by LDA configurations for feature location. 70
3.12 Box plots of the effectiveness measure for feature location for ArgoUML and jEdit. 71
3.13 Variability of performance achieved by LDA configurations for labeling. . . . 74

4.1 Building a defect prediction model. 81
4.2 Graphical interpretation of Pareto dominance. 89
4.3 Decision tree for defect prediction. 91
4.4 Performances of predicting models achieved when optimizing inspection cost and number of defect-prone classes. 100
4.5 Performances of predicting models achieved when optimizing inspection cost and number of defect-prone classes. 101
4.6 Performances of predicting models achieved when optimizing inspection cost and number of defects. 104
4.7 Performances of predicting models achieved when optimizing inspection cost and number of defects. 105

5.1 Example of triangle classification program 121

5.2 Example of flag landscape. The y-axis measures the branch coverage while the x-axis represents the input space. 123
5.3 Objective function landscape by Pargas *et al.* [20] for a simple program. . . . 127
5.4 Objective function landscape by Wegener *et al.* [21] for a simple program. . . 128

6.1 Geometrical interpretation of SVD applied to a population P aiming at estimating evolution directions. 143
6.2 NSGA-II vs. SVD-NSGA-II on MOP test functions 157
6.3 NSGA-II vs. SVD-NSGA-II on shifted ZDT6 when $n = 50$. 158
6.4 NSGA-II vs. SVD-NSGA-II on DTLZ1 and DTLZ2 when $n = 10$. 160
6.5 NSGA-II vs. SVD-NSGA-II on DTLZ3, DTLZ4 and DTLZ6 when $n = 50$. . . 161

7.1 Objective function landscape by Wegener *et al.* [21] for the program *example*1. 165
7.2 Objective function landscape by Wegener *et al.* [21] for the program *example*2. 166
7.3 An example of class under test 170
7.4 Example of unreachable branch for *Sort*. 178
7.5 Comparison of St-GA, H-OV-GA, OV-GA and R-GA over on QRDecomposition. 179
7.6 Comparison of the budget consumed by St-GA, H-OV-GA, OV-GA and R-GA over 100 independent runs on Sort. 182

8.1 Graphical representation of the orthogonal arrays $L_4(2^3)$ for three test cases. 189
8.2 Hypervolume metric based on *cost* and *effectiveness* of the sub-test suites. . . 200
8.3 Pareto frontiers. 204
8.4 Three-objective Pareto Frontiers achieved on *flex*. 207
8.5 Three-objective Pareto Frontiers achieved on *gzip*. 207
8.6 Three-objective Pareto Frontiers achieved on *printtokens*. 208
8.7 Effectiveness of the achieved sub-test suites for two-objective test case selection. 210
8.8 Effectiveness of the achieved sub-test suites for three-objectives test case selection. 211

A.1 NSGA-II vs. SVD-NSGA-II on MOP2 test problem when $n = 100$ ($n = 50$ is reported in Chapter 6). 225
A.2 NSGA-II vs SVD-NSGA-II on ZDT3 test problem when $n = 100$ ($n = 50$ is reported in Chapter 6). 225
A.3 NSGA-II vs SVD-NSGA-II on ZDT6 test problem when $n = 100$ ($n = 50$ is reported in Chapter 6). 226
A.4 NSGA-II vs. SVD-NSGA-II on MOP1 problem. 226
A.5 NSGA-II vs SVD-NSGA-II on shifted ZDT1 test problem. 227
A.6 NSGA-II vs. SVD-NSGA-II on shifted ZDT2 test problem. 227
A.7 NSGA-II vs SVD-NSGA-II on shifted ZDT4 test problem. 227
A.8 NSGA-II vs. SVD-NSGA-II on DTLZ1 when $n = 50$. 228
A.9 NSGA-II vs. SVD-NSGA-II on DTLZ1 when $n = 100$. 228

A.10 NSGA-II vs. SVD-NSGA-II on DTLZ2 when $n = 50$. 228
A.11 NSGA-II vs. SVD-NSGA-II on DTLZ2 when $n = 100$. 229
A.12 NSGA-II vs. SVD-NSGA-II on DTLZ3 when $n = 10$. 229
A.13 NSGA-II vs. SVD-NSGA-II on DTLZ4 when $n = 50$. 229
A.14 NSGA-II vs. SVD-NSGA-II on DTLZ4 when $n = 100$. 230
A.15 NSGA-II vs. SVD-NSGA-II on DTLZ6 when $n = 50$. 230
A.16 NSGA-II vs. SVD-NSGA-II on DTLZ6 when $n = 100$. 230

C.1 Pareto frontiers achieved on *bash*, *flex*, *grep* and *gzip* for two-objectives test suite optimization problem 236
C.2 Pareto frontiers achieved on *printtokens*,*printtokens2*, *schedule*, *schedule* and *sed* for two-objectives test suite optimization problem 237
C.3 Pareto frontiers achieved on *space* and *vim* for two-objectives test suite optimization problem 238
C.4 Three-objective Pareto Frontiers achieved on *bash* 238
C.5 Three-objective Pareto Frontiers achieved on *flex*. 238
C.6 Three-objective Pareto Frontiers achieved on *grep* 239
C.7 Three-objective Pareto Frontiers achieved on *gzip*. 239
C.8 Three-objective Pareto Frontiers achieved on *printtokens*. 239
C.9 Three-objective Pareto Frontiers achieved on *printtokens2* 240
C.10 Three-objective Pareto Frontiers achieved on *schedule* 240
C.11 Three-objective Pareto Frontiers achieved on *schedule2* 240
C.12 Three-objective Pareto Frontiers achieved on *sed* 241
C.13 Three-objective Pareto Frontiers achieved on *space* 241
C.14 Three-objective Pareto Frontiers achieved on *vim* 241
C.15 Effectiveness of the achieved sub-test suites for two-objective test case selection on *bash*, *flex*, *grep* and *gzip*. 242
C.16 Effectiveness of the achieved sub-test suites for two objectives test case selection on *printtokens*,*printtokens2*, *schedule*, *schedule*, *sed* and *space*. 243
C.17 Effectiveness of the achieved sub-test suites for two objectives test case selection on *vim*. 244
C.18 Effectiveness of the achieved sub-test suites for three objectives test case selection *bash*, *flex*, *grep* and *gzip*. 245
C.19 Effectiveness of the achieved sub-test suites for three objectives test case selection on *printtokens*,*printtokens2*, *schedule*, *schedule*, *sed* and *space*. 246
C.20 Effectiveness of the achieved sub-test suites for three objectives test case selection on *vim*. 247

List of Tables

3.1 Values of the genes (steps of the IR process) for LSI-GA. 49
3.2 Characteristics of the systems used for traceability recovery. 52
3.3 Comparison of traceability recovery performances: average precision values. . 56
3.4 Comparison of traceability recovery performances (precision): results of the Wilcoxon test. 56
3.5 Comparison of different IR processes provided by LSI-GA, ideal and reference. 57
3.6 Task 2 (feature location): characteristics of the datasets used in the experiment. 57
3.7 Comparison of feature location performances (EM): the results of the Wilcoxon test. 59
3.8 Comparison of different IR processes provided by LSI-GA, ideal and reference. 59
3.9 Comparison of bug report duplication: the results of the Wilcoxon test. . . . 62
3.10 Comparison of different IR processes provided by LSI-GA, ideal and reference on Eclipse. 63
3.11 Characteristics of the systems used for traceability recovery. 64
3.12 Results of the Wilcoxon test for traceability recovery. 68
3.13 Characteristics of the systems used in feature location. 68
3.14 Results of the Wilcoxon test on feature location performances. 71
3.15 Characteristics of the systems used in source code labeling. 72
3.16 Average Overlap between automatic and manual labeling. 75

4.1 Java projects used in the study. 94
4.2 MODEP vs. single-objective predictors when optimizing inspection cost and number of defect-prone classes identified. Single-objective predictors are also applied using a within-project training strategy. 102
4.3 MODEP vs. single-objective predictors when optimizing inspection cost and number of defects identified. Single-objective predictors are also applied using a within-project training strategy. 106
4.4 MODEP vs. local predictors when optimizing inspection cost and number of defect-prone classes predicted. 107
4.5 MODEP vs. local predictors when optimizing inspection cost and number of defects predicted. 108

4.6 The first 10 Pareto optimal models obtained by MODEP (using logistic regression) on *Log4j*. 110

5.1 Tracey's objective functions for relational predicates. k can be any arbitrary positive constant value. 126

6.1 Single-objective test functions used in our numerical experiment. The functions $f_7, f_8, f_9, f_{14}, f_{15}$ come from CEC 2008 Special Session [22]. 149
6.2 Function error values achieved on multimodal test functions by GA, SVD-GA, and CMA-ES when n=50. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference. . . 151
6.3 Function error values achieved on unimodal test functions by GA, SVD-GA, and CMA-ES when n=50. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference. . . 152
6.4 Two-objectives test problems used in our numerical experiment. The number of variables is $n = 50,\ 100$. 154
6.5 Three-objectives test problems used in our numerical experiment. 154
6.6 IGD values achieved by SVD-NSGA-II and NSGA-II for n=50. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference. 156
6.7 IGD values achieved by SVD-NSGA-II and NSGA-II on the DTLZ test suite. Values are shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference. 159

7.1 Subjects used in our study. 172
7.2 Average coverage values achieved by St-GA, R-OV-GA, H-OV-GA and R-GA over 100 independent runs. Values in bold face turned out to be the best against those achieved by the other algorithms for the same CUT. 177
7.3 P-values achieved using Wilcoxon Rank Sum test with Holm's correction and Cohen d effect sizes. We use S, M, and L to indicate small, medium and large effect sizes respectively. 180
7.4 Average budget consumed by St-GA, R-OV-GA, H-OV-GA and R-GA over 100 independent runs (the percentage variations with respect to St-GA are also shown). Values in bold face turned out to be the best (lowest) against those achieved by the other algorithms for the same CUT. 181
7.5 P-values achieved using Wilcoxon Rank Sum test with Holm's correction and Cohen d effect sizes. Also, we use S, M, and L to indicate small, medium and large effect sizes respectively. 181

8.1 Orthogonal arrays $L_4(2^3)$ for three test cases where there are four combinations of test cases (factors). The row vectors of such a matrix $L_4(2^3)$ can be used as an initial binary population for GAs. 189

8.2 Programs used in the study. 192
8.3 Configurations of NSGA-II and DIV-GA for the programs used in the study. 197
8.4 Two-objective test case selection: mean Pareto sizes and number of non-dominated solutions achieved by the different algorithms. 202
8.5 Comparison between different algorithms for the two-objective test case selection problem. Welch's t-test p values and Cohen's d effect size. We use S, M, and L to indicate small, medium and large effect sizes, respectively. 203
8.6 Three-objective formulation of the test case selection problem: mean size of Pareto frontier and mean number of non-dominated solutions obtained by the different algorithms. 205
8.7 Comparison between different algorithms for the three-objective test case selection problem. Welch's t-test p values and Cohen's d effect size. We use S, M, and L to indicate small, medium and large effect sizes respectively. 206
8.8 Mean cost-effectiveness hypervolume values. 209

A.1 Function error values achieved on multimodal test functions by GA, SVD-GA, and CMA-ES when n=100. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference. . . 222
A.2 Function error values achieved on unimodal test functions by GA, SVD-GA, and CMA-ES when n=100. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference. . . 223
A.3 IGD values achieved by SVD-NSGA-II and NSGA-II for n=100. Values are shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference. 224

Acronyms

CBO Coupling Between Objects

CFG Control Flow Graph

CK Chidamber/Kemerer software metric suite

CUT Class Under Test

CMA-ES Covariance Matrix Adaptation Evolution Strategy

DIT Depth of Inheritance

DSMs Design Structure Matrices

EM Effectiveness Measure for feature location

EVM EValuation Metric

GA Genetic Algorithm

GP Genetic Programming

GPGPU General Purpose Graphical Processing Unit

H-OV-GA GA with History aware Orthogonal exploration

IR Information Retrieval

LCOM Lack of Cohesion Among Methods

LDA Latent Dirichlet Allocation

LSI Latent Semantic Indexing

LOC Lines of Codes

MLP Multi-Layer Perceptron

MOGA Multi-Objective Genetic Algorithm

NOC Number Of Children

NPGA Niched Pareto Genetic Algorithm

MQ Modularization Quality

NSGA Non-dominated Sorting Genetic Algorithm

NSGA-II Elitist Non-dominated Sorting Genetic Algorithm

RBF Radial Basis Function

RFC Response For a Class

R-GA GA with Reactive exploration

R-OV-GA GA with both Orthogonal and Reactive exploration

RMSE Root-Mean-Square Error

RTM Relational Topics Model

SBSE Search-Based Software Engineering

SBST Search-Based Software Testing

SPEA Strength Pareto Evolutionary Algorithm

SUT System Under Test

SVD Singular Value Decomposition

SVD-GA Genetic Algorithm based on Singular Value Decomposition based

SVD-NSGA-II Elitist Non-dominated Sorting Genetic Algorithm based on Singular Value Decomposition

SVR Support Vector Regression

TCS Test Case Selection

TSM Test Suite Minimization

TCP Test Case Prioritization

VEGA Vector Evaluated Genetic Algorithm

VSM Vector Space Model

WMC Weighted Methods per Class

Chapter 1

Introduction

Contents

1.1 Research problem and motivations 2

1.2 Major Contributions 4

1.2.1 IR-based program comprehension using search algorithm 4

1.2.2 Multi-objective defect prediction 5

1.2.3 Improving search-based software testing through diversity injection 6

1.2.4 Improving test suite optimization through diversity injection . . . 7

1.3 Thesis organization 7

1.1 Research problem and motivations

In software engineering there are many expensive tasks that are performed during development and maintenance activities. Therefore, there has been a lot of effort to try to automate these tasks in order to significantly reduce the development and maintenance cost of software, since the automation would require less human resources. One of the most used way to make such an automation is the *Search-Based Software Engineering* (SBSE), which reformulates traditional software engineering tasks as search problems [23]. In SBSE the set of all candidate solutions to the problem defines the search space [24] while a fitness function differentiates between candidate solutions providing a guidance to the optimization process [23]. After the reformulation of software engineering tasks as optimization problems, search algorithms are used to solve them. Several search algorithms have been used in literature [25], such as genetic algorithms [26, 27, 28], genetic programming [29, 30, 31], simulated annealing [32], hill climbing (gradient descent) [33], greedy algorithms [34, 35, 36], particle swarm [37] and ant colony [38].

The earliest work on the application of search-based optimization to software engineering was proposed by Miller and Spooner in 1976 [39] for software testing and later Chang [40] promoted the application of evolutionary algorithms for software management problems in 1994. The term SBSE was coined later by Harman and Jones [23] in 2001 and nowadays it includes all research works coming from many and different areas of software engineering, other than software testing and software management plan. SBSE has enough material to justify the publication of several surveys on SBSE [41, 42, 43, 25, 44] and search-based software testing (SBST) [27, 45, 46]. As revealed by a survey [44], 54% of the overall SBSE literature is related to software testing [47, 48, 49, 50]. However, SBSE has been applied to many software engineering problems such as requirement engineering [51], project planning and cost estimation [52, 53, 54, 31], automated maintenance [32, 55, 56], service-oriented software engineering [57], compiler optimisation [58] and quality assessment [32]. SBSE is not only an academic research area, but it has been widely used in industry in many industrial applications. For example, Microsoft used search-based techniques within its PeX software testing tool in order to automatically test software programs with floating-point input data and constraints [59, 60] while Google used multi-objective optimization techniques for regression testing into its software development life cycle [61]. NASA [62] Motorola [63] and Ericsson [64] used search algorithms for requirements analysis and optimization.

As pointed out by many previous works [41, 42, 43, 25, 44], SBSE is widely applicable to many software engineering problems and it also provides to these problems several advantages [41]:

- *Generality.* SBSE is applicable in all software engineering tasks that can be formulated as optimization problems. It also provides a way to deal with hard, highly constrained problems that involve many conflicting objectives using an automated approach. It is also likely that there is a suitable fitness function with which one could start experimentation since many software engineering metrics are readily exploitable as fitness

functions [25].

- *Robustness.* Search based optimization techniques are stochastic algorithms that can be tuned using a selection of parameters. However, it should be noted that such algorithms are extremely robust and that often the solutions require only lie within some specified tolerance [25].

- *Scalability.* Many approaches that are sound in the laboratory, turn out to be inapplicable in the field, because they lack scalability [65]. Fortunately, some search algorithms are parallel, since they use multiple solutions to explore many regions of the search space, such as genetic algorithms. This intrinsic parallelism can be used with distributed computation [66, 33] to increase the scalability of SBSE approaches. For example recent work has also shown how General Purpose Graphical Processing devices (GPGPU) can be used to scale the computation of a search process, showing better performance than CPU-based computation [65].

- *Re-unification.* It is possible to identify links between apparently unconnected software engineering problems by looking them from an optimization point of views [44]. For instance, regression testing and requirements engineering are two software engineering areas that are typically involved in two opposite steps of software life cycle. However, regression and requirements optimization problems share common formulations. Viewed as optimization problems, they essentially consist in selection and prioritization problems and they can be solved with similar search-based approaches.

For these reasons, this thesis investigates and proposes the usage of search based approaches to reduce the effort of software maintenance and software testing with particular attention to four main activities: (i) program comprehension; (ii) defect prediction; (iii) test data generation and (iv) test suite optimization for regression testing. For program comprehension and defect prediction, this thesis provided their first formulations as optimization problems and then proposed the usage of genetic algorithms to solve them. More precisely, this thesis investigates the peculiarity of source code against textual documents written in natural language and proposes the usage of Genetic Algorithms (GAs) in order to calibrate and assemble IR-techniques for different software engineering tasks. This thesis also investigates and proposes the usage of Multi-Objective Genetic Algorithms (MOGAs) in order to build multi-objective defect prediction models that allows to identify defect-prone software components by taking into account multiple and practical software engineering criteria.

Test data generation and test suite optimization have been extensively investigated as search-based problems in literature [67, 68]. However, despite the huge body of works on search algorithms applied to software testing, both (i) automatic test data generation and (ii) test suite optimization present several limitations and not always produce satisfying results [69, 68]. The success of evolutionary software testing techniques in general, and GAs in particular, depends on several factors. One of these factors is the level of diversity among the individuals in the population, which directly affects the *exploration* ability of the search.

For example, evolutionary test case generation techniques that employ GAs could be severely affected by *genetic drift*, i.e., a loss of diversity between solutions, which lead to a premature convergence of GAs towards some local optima. For these reasons, this thesis investigate the role played by diversity preserving mechanisms on the performance of GAs and proposed a novel diversity mechanism based on Singular Value Decomposition and linear algebra. Then, this mechanism has been integrated within the standard GAs and evaluated for evolutionary test data generation. It has been also integrated within MOGAs and empirically evaluated for regression testing.

Next section provides a summary of the main contributions of this thesis.

1.2 Major Contributions

This section presents the research contributions of this thesis derived from the analysis of the research problems briefly described in the previous section.

1.2.1 IR-based program comprehension using search algorithm

Extracting, representing, and analyzing conceptual information in software documents represent core activities for understanding the software to maintain. During program comprehension, developers read the source code in order to build a *cognitive model* that is used to form a mental representation of the program to be understood and changed [70]. Such a cognitive process could be tedious, error prone and time consuming in large software systems, where the developer is requested to read (and comprehend) a large number of source code lines. In such a scenario, developers spend more time reading and navigating the code than writing it [71, 72].

Information Retrieval (IR) methods were proposed and used to support software engineers during comprehension tasks [73, 74, 75, 76, 77]. All IR-based techniques that support SE tasks, such as Latent Semantic Indexing (LSI) [78] or Latent Dirichlet Allocation (LDA) [79], require configuring different components and their respective parameters, such as type of pre-processors (e.g., splitting compound or expanding abbreviated identifiers; removing stop words and/or comments), stemmers (e.g., Porter or snowball), indexing schemata (e.g., term frequency - inverse document frequency), similarity computation mechanisms (e.g., cosine, dot product, entropy-based), etc. Even though IR techniques was successfully used in the IR and natural language analysis community, applying it on software data, using the same parameter values used for natural language text, did not always produce the expected results [80]. A poor parameter calibration or wrong assumptions about the nature of the data could lead to poor results [81]. Recent research has challenged this assumption and showed that text extracted from source code is much more repetitive and predictable as compared to natural language text [82]. This also makes the practical use of IR-based processes quite difficult and undermines the technology transfer to software industry.

This thesis starts from the findings that IR techniques should not be applied as done

in traditional information retrieval problems because text in software artifacts has different properties, as compared to natural language text. For these reasons, this thesis considers a novel approach to solve the problem of assembling IR-based solutions for a given SE task and accompanying dataset. The main assumption, which is supported by a large body of empirical research in the field [80], is that it is not possible to build a set of guidelines for assembling IR-based solutions for a given set of tasks as some of these solutions are likely to under-perform on previously unseen data-sets. The solution proposed in this thesis consists of (i) formulating of the problem of finding the best IR configuration as an optimization problem, and (ii) using genetic algorithm to find the (near) optimal configuration. The approach is unsupervised and task-independent because it is based on the quality of the clustering of the indexed software artifacts and not on past data. Thus, it can be used to select and generate on demand an adequate IR-based solution given a dataset provided as input.

1.2.2 Multi-objective defect prediction

Effective *defect prediction* models can help developers to focus activities such as code inspection or testing on likely defect-prone components, thus optimizing the usage of resources for quality assurance. The idea is that software repositories provide information about past faulty software entities that can be used to training data mining and machine learning algorithms to predict and locating defects of future entities. A good defect prediction model should be able to identify the largest proportion of defect-prone components (effective) and should limit the number of false positives (efficient), thus not wasting the developers' effort in the quality assurance task. However, building effective and efficient bug prediction model requires the availability of enough accurate data about a project and its history [83]. This clearly limits the adoption of prediction models on new projects. Several authors [84], [85], [86], [87] have discussed the possibility to use data from other projects to train machine learning models to deal with the limited availability of training data. This strategy, referred to as *cross-project defect prediction*, does not always produce the expected performances [84, 85].

An important limitation of defect prediction models for cross-project defect prediction is that they are built using the same algorithms and the same performance metrics that are used for traditional classification problems, such as the number of defective software components erroneously classified. Recently, researchers [88, 89, 90] have pointed out that defect prediction models should be evaluated using other performance metrics, not used for traditional classification problems, in order to take into account goals and needs of the software engineer, such as inspection cost or defect density. In this scenario, a good model is the one that identifies defect-prone files with the best ratio between (i) effort spent to inspect such files, and (ii) number of defects identified. However, even if previous studies considered cost-effectiveness to assess the quality of a predictor [88, 89, 90], they still built prediction models using the same algorithms used for traditional classification problems which, by definition, find the model that minimizes the fitting error, precision and recall without taking into account (i) inspection cost and (ii) defect density.

Stemming from the considerations by previous study [88, 89, 90], it can be argued the ML techniques should not be applied, built and evaluated as in traditional classification problems. Defect prediction models should take into account not only the accuracy of prediction but also other metrics that have practical relevance for software engineer such as the inspection cost and the defects density of software components. This thesis provides the first explicit formulation of the problem of finding cost-effect prediction models as a multi-objective optimization problem. Then, evolutionary algorithm, and more precisely MOGAs, are used to find (near) optimal solutions for this problem.

1.2.3 Improving search-based software testing through diversity injection

The first studied research area on SBSE is the automated test data generation. Despite the huge body of works on this area, automatically testing the 100% of code is still an open issue because some branches of code can be particular complex and hard to cover using search-algorithms. The success of GAs depends on the level of diversity among the individuals in the population, which directly affects the *exploration* ability of the search. Indeed, given the fact that the search space is usually extremely large, the GA has to maintain an adequate level of diversity in the population in order to effectively explore the search space and look for alternative, potential solutions. However, traditional genetic operators might not suffice by themselves in maintaining enough diversity. Depending on the difficulty of the current search target and the type of selection scheme in use, individuals in the population can become too similar to one another, eventually converging to a single, sub-optimal individual. Hence, the ability of the search to explore new areas of the search space is greatly reduced. This phenomenon is referred to as *genetic drift* [91]. Hence, test data generation techniques need to properly address this problem if they are to succeed on programs with complex logic and branches. Previous work treated the problems of diversity considering simple heuristics promoting diversity between individuals within the same generation [92, 93]. Hence, new offsprings might explore regions that have been explored in previous generations while new potential regions can still remain unexplored, increasing the likelihood to reach sub-optimal solutions.

This thesis investigates the role of diversity for genetic algorithms and integrates in the standard GA an orthogonal exploration mechanism of the search space through the estimation of the evolution directions via Singular Value Decomposition (SVD) with the aim at augmenting the population diversity. Results achieved on 17 Java classes extracted from well-known libraries show that with the appropriate application of diversification techniques, the effectiveness and efficiency of GAs in structural test data generation can be greatly improved. In particular, effectiveness (coverage) was significantly improved in 47% of the subjects and efficiency (search budget consumed) was improved in 85% of the subjects on which effectiveness remains the same.

1.2.4 Improving test suite optimization through diversity injection

Re-testing the whole software system by executing all the available test cases might be too expensive and unfeasible, especially for large systems [94, 95]. The problem is clearly amplified by the growth of the test suites as the system evolves. This has led to develop of strategies in order to optimize test suite according to some testing criteria to be satisfied. For example, widely used criteria are code coverage [34, 96, 97], program modification [98, 99, 100], execution cost [68, 101, 102], or past fault information [97, 68, 103]. The test suite optimization problem has been also formulated as a combination of multiple—often contrasting—criteria. Results have highlighted that when using multiple criteria the optimization of test suite is more effective than when using individual ones [97, 104, 68, 103, 105]. Hence, test suite optimization has been treated using MOGAs to deal with multiple and contrasting objectives [68, 103]. Empirical results indicated that in some cases MOGAs provide better performance than other single-objective algorithms. However, there is no a clear winner between single-objective approach and MOGAs [68] and their combination is not always useful to achieve better results [103]. The poor performance of MOGAs when applied to test suite optimization can be due to the phenomenon of *genetic drift*, i.e., a loss of diversity between solutions, which affects not only single-objective GAs but also MOGAs. Thus, promoting diversity between test cases is a key factor to improve the optimality of MOGAs [106].

This thesis investigates the role of diversity for MOGAs and integrates in the main loop of one the most popular MOGAs, called NSGA-II [107], two novel genetic operators to promote diversity between the selected test cases without adding any further diversity-based objective function: (i) a generative algorithm to build a diversified initial population, based on *orthogonal design* [108], and (ii) an orthogonal exploration mechanism of the search space, trough Singular Value Decomposition (SVD) [109], aimed at preserving the diversity during the evolution of the population. These two diversity mechanisms can be applied for any test suite optimization problem and independently of the number of test criteria or objectives to be taken into account. The thesis investigates the usefulness of the proposed diversity mechanisms through an empirical study conducted on 11 real world open-source programs. Results show that the proposed algorithm outperforms both traditional MOGAs and greedy algorithms. The thesis also presents a performance metric, inspired by the traditional hypervolume metric widely used for numeric multi-objective problems [110], to evaluate the ability of the selected test cases to reveal faults (effectiveness) in a multi-objective paradigm.

1.3 Thesis organization

This thesis is composed of 9 chapters including this chapter. In particular, the thesis is composed of two parts:

- **Part I**: this part considers two software maintenance activities, that are (i) IR-based program comprehension and (ii) defect predictions. It also provides a search-based

formulation for both the two activities. Specifically, Chapter 2 gives an overview on optimization problems, GAs, MOGAs and discusses related work on search-based software maintenance. Chapter 3 contains (i) the formal formulation of finding the best the best IR process as an optimization problem; (ii) the description of a novel a search-based approach based on GAs to solve such a problem; and (iii) an empirical evaluation of the proposed approach through a case study involving different software engineering tasks. This chapter also shows how using GAs to calibrate sophisticate IR methods such as LDA. Finally, Chapter 4 provides (i) the formal formulation of finding cross-project defect prediction models as a multi-objective optimization problem; (ii) the definition of a novel a search-based approach based on MOGAs to solve such a problem; and (iii) an empirical evaluation of the proposed approach through a case study.

- **Part II**: this part presents the contribution of the thesis within the context of software testing. It is composed of 4 chapters. Chapter 5 presents discusses related work on evolutionary test data generation, test suite optimization and their limitations. Chapter 6 presents the novel diversity preserving techniques proposed in this thesis aimed at improving the performance of both GAs and MOGAs when applied to complex problems by reducing the problem of *genetic drift*. This chapter also presents a preliminarily evaluation of the proposed techniques on numeric test benchmark problems widely used in evolutionary computation community to test the performance of search algorithms. Chapter 7 presents the empirical evaluation of diversity preserving techniques proposed in Chapter 6 and its variants, when integrated into the main loop of GAs for evolutionary test data generation. Chapter 8 presents an empirical evaluation of diversity preserving techniques proposed in Chapter 6 and its variants, when integrated into the main loop of MOGAs for solving multi-objective test suite optimization problem.

Finally, Chapter 9 gives conclusion remarks and directions for future work.

Part I

Software Maintenance

Chapter 2

Background: Genetic Algorithms

Contents

2.1 Introduction and Motivations . . . 12
2.2 Search-based approaches to Software Maintenance . . . 13
2.3 Optimization problems . . . 15
2.4 Evolutionary Computation . . . 17
2.5 Single-objective Genetic Algorithms . . . 19
2.5.1 Solution encoding . . . 20
2.5.2 Selection operators . . . 21
2.5.3 Crossover operators . . . 22
2.5.4 Mutation operators . . . 24
2.5.5 Elitism . . . 25
2.5.6 Termination . . . 25
2.6 Multi-objective Genetic Algorithms . . . 26
2.6.1 NSGA-II . . . 27

2.1 Introduction and Motivations

During the past few decades, there has been a rapid increase of software systems to support humans in several activities including health-care services, manufacturing industries, financial company, etc. Software systems are dynamic entities subjected to continuous and frequent changes and corrective operations that are generally needed to support, to preserve or to improve software quality, such as functionality, flexibility, correctness, reliability, etc. A change can involve small parts of the software system, such as changes performed to fix errors discovered in the system, or a more complex and massive operation aimed at re-engineering the system to improve its performances [111]. According to the IEEE Standard the *process of modifying a software system or its components after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment*[1] is defined as *software maintenance*. Maintenance is needed to ensure that the software continues to satisfy user requirements. However, a system always evolves through maintenance activities to meet the requirements of the changing world in which it operates [112], and this makes changes hard, complex and expensive. Software maintenance involves the execution of many complex operations other than the simple source code editing [111], such as (i) planning phase, (ii) understanding and comprehension of the source code and its behavior; (iii) identification of the components to be modified; (iv) impact analysis; (v) change implementation; (vi) testing of new components implemented during software maintenance; (vii) regression testing aimed at verifying whether new changes have introduced errors into unchanged parts, endangering their behaviors [27].

According to various studies [113, 114], software maintenance activities consume about the 60% and the 90% of the cost of the entire software system life cycle. Thus, companies spend the majority of time and resources to maintain and change existing software than to design and develop new software. Maintenance activities that are too expensive can have practical implication for software companies because they can negatively affect the capabilities to release new version of existing systems. Researches also reported that the majority of software maintenance resources are devoted to program comprehension [115, 71, 72], software verification and validation [116] rather than implementing changes. For these reasons, this first part of the thesis focuses on two critical activities performed before/during maintenance activities: program comprehension and defect prediction. To support program comprehension many researchers have proposed the usage of IR-based techniques, such as Latent Semantic Indexing (LSI) [78] or Latent Dirichlet Allocation (LDA) [79], to automatically extract and manage textual information from source code and software documentation. As pointed out by previous works [81, 80], these techniques require to accurately configure different components and parameters in order to successfully support the software engineers. Moreover, the problem of building a set of guidelines for assembling IR-based solutions for a given dataset or for a specific task is impossible [80], because different datasets and different software engineering tasks requires different parameters with different calibrations of IR

[1]Standard ISO-IEC/IEEE 14764-2006.

methods. To address these issues, in this thesis we propose the usage of search based techniques to automatically calibrate IR methods. Therefore, we provide the first formulation of the problem of calibrating IR-techniques for SE tasks as an optimization problem and then we suggest to use optimization algorithm to solve it.

Machine Learning (ML) techniques have been used to identifying likely fault-prone software components [88, 83, 84, 85, 86], in order to prioritize Quality Assurance (QA) activities in the presence of limited time or resources available, thus, reducing the effort of software verification and validation. An important limitation of defect prediction models for cross-project defect prediction is that they are built using the same algorithms and the same performance metrics that are used for traditional classification problems, such as the number of defective software components erroneously classified. Recently, researchers [88, 89, 90] have pointed out that predicting defect prone software components is not equivalent to traditional classification problems, since for software engineer is more useful to consider inspection cost or defect density, instead of the prediction error. However, even if previous studies considered cost-effectiveness to assess the quality of a predictor [88, 89, 90], they still built prediction models using the same algorithms used for traditional classification problems which, by definition, find a model that minimizes the fitting error without taking into account (i) inspection cost and (ii) defect density. To address these issues, in this thesis we propose the usage of search based techniques to build defect prediction models which take into account not only the accuracy of prediction but also other metrics that have practical relevance for the software engineer. Specifically, this thesis provides the first explicit formulation of the problem of finding cost-effect prediction models as a multi-objective optimization problem. Then, evolutionary algorithm, and more precisely MOGAs, are used to find (near) optimal solutions for this problem.

The rest of the chapter is organized as follows: Section 2.2 summarizes related work on search-based approaches to software maintenance. Section 2.3 provides background notions on optimization problems, while Section 2.4 describes evolutionary algorithms and GAs in particular. Finally, Section 2.5 and Section 2.6 summarize background notions about single objective genetic algorithms and multi-objective genetic algorithms.

2.2 Search-based approaches to Software Maintenance

According to a recent survey on SBSE [44], the majority of search-based works on software maintenance focused on software modularization and refactoring. The earliest work on search-based software modularization was proposed by Mancoridis *et al.* in 1998 [117] and presented a tool called *Bunch* that implements clustering techniques to cluster software modules. In this work, the quality of software modularization is measured using a single fitness function, named *Modularization Quality* (MQ), which condenses two contrasting goals: (i) maximizing cohesion and (ii) minimizing coupling [55]. Harman *et al.* [55] used evolutionary algorithms to optimize a fitness function that integrates module granularity into MQ. Mahdavi *et al.* [33] described a multiple hill climbing approach for optimizing the MQ fitness

function. In this approach an initial set of hill climbs is performed and from these, a set of best hill climbs is identified according to some threshold. Harman *et al.* [118] empirically compared the robustness of two fitness functions used for software module clustering: (i) the MQ function used by previous works and (ii) EValuation Metric (EVM).

Search-based software clustering techniques have been also used for further applications in SE applications, other than the original work on software modularization. Bodhuin *et al.* [119] proposed to cluster together (in jars) classes that, for a set of usage scenarios, are likely to be used together. They used genetic algorithms to minimize a fitness function that considers the packaging size and the average downloading times. Huynh and Cai [120] proposed an automated approach to check the consistence between source code modularity and designed modularity. Their approach uses design structure matrices (DSMs) as a uniform representation; it uses existing tools to automatically derive DSMs from the source code and design, and uses a genetic algorithm to automatically cluster DSMs and check the consistence. Cohen [121] used genetic algorithms in order to cluster multi-threaded applications that use single shared heap memory in order to improve garbage collector operations. Del Rosso [122] proposed the usage of genetic algorithms in order to improve the internal memory fragmentation by finding the optimal configuration of a segregated free lists data structure.

Other works proposed search-based approaches in order to automatically change the structure of software programs, in order to improve some software quality metric, without altering the behavior and the semantics of the programs. For example, O'Keefee and Cinnéide [123] presented a software tool capable of refactoring object-oriented programs to conform more closely to a given design quality model, by formulating the task as a search problem in the space of alternative designs. Harman and Tratt [124] proposed the usage of Pareto efficient multi-objective evolutionary algorithms to suggest a sequence of refactoring steps that are (near) optimal according to multiple software quality metrics. Bouktif *et al.* [125] proposed to adopt meta-heuristic approaches to schedule refactoring operations in order to remove duplicated code under constraints and priority. Other works [126, 125, 127] proposed the usage of search-based approaches to improve the performance of compiled code by searching the space of compiler options. For example, Dubach *et al.* [125] considered 60 different optimizations for the `gcc` compiler and used as objective functions the compilation time and code quality (measured in terms of execution time). Fatiregun *t al.* [128] and Kessentini *et al.* [129] proposed program transformations via search-based approaches to reduce programs size and to automatically construct amorphous slices. Nisbet [130] proposed a framework based on GAs to find the optimal sequence of code transformations that minimizes the execution time of loop-based programs written in FORTRAN for parallel architectures.

Other than software modularization and refactoring, previous works considered other SBSE applications that do not fall in these two categories. For example, Bate and Emberson [131] incorporated scenario based analysis into heuristic search strategies to improve the flexibility of real-time embedded systems. Saharaoui *et al.* [132] presented two search-based algorithms for identifying objects in procedural code and facilitating the migration of legacy systems to object-oriented technology. Forrest *et al.* [133] used genetic programming

combined with program analysis methods to repair bugs in off-the-shelf legacy C programs.

In summary, all previous works on search-based software maintenance focused on automatic modifications of programs in order to improve some software quality metrics (such as cohesion, coupling, memory usage, etc.) without modifying their behavior. All these approaches apply automatic changes in order to improve performance, functionality or in order to reduce the effort of future maintenance activities. To the best of our knowledge, no previous work rigorously focused on using search-based approaches to automatically calibrate IR processes/methods to support program comprehension or for building multi-objective defect prediction models. In this thesis we proposed the first formulation of these two problems as optimization problems, and then we propose the usage of genetic algorithms to solve/optimze them. Since for both program comprehension and defect prediction we use search-based approaches, the next sections provides background notions on optimization problems and search algorithms, with particular attention to genetic algorithms.

2.3 Optimization problems

According to [23] there are three key ingredients to use in order to reformulate a software engineering task as an optimization problem: (i) a representation of the problem which allows the symbolic manipulation; (ii) a fitness function that captures the objective or objectives to be optimised; (iii) a set of manipulation operators which allow to change candidate solutions. The *representation* of the candidate solutions is a critical phase of any search problem and it depends on the nature of the problem itself. Representations that are widely used for problem parameters are floating point and binary code [23]. The *fitness function* allows to characterize what is a *good* solution to a given problem. It defines the fitness landscape, i.e., it assigns an ordinal scale of goodness to the solutions it is applied to. Such a landscape should be not too flat, nor should has too local optima that can prevent the identification of a global optimum to the problem [25]. However, this aspect is highly related to the nature of the problem being solved with SBSE approaches. Some *operators* are used to mutate candidate solutions to produce other nearby candidate solutions. Different search algorithms differ on the number and the type of operators.

An *optimization problem* (or *search problem*) is the problem of finding the best solution from the set of all feasible solutions according to a given function to be optimized. It can be of different kinds on the basis of the number goals to be considered in the search task. The optimization problem can have only one objective function to be optimized and in this case the problem is named *single-objective optimization problem.* In other cases the optimization problem can involve multiple and conflicting objective functions to be optimized in the same time and in this case it is named *multi-objective optimization problem.* Some problems can have only one optimum (*unimodal problem*) while other may contain more than one optimum in the search space (*multimodal problem*). The standard definition of a single-objective optimization problem is the following:

Definition 1. *Given an objective function* $f : \Omega \rightarrow \mathbb{R}$*, where* Ω *is the set of all possible*

solutions. Find the solution (or decision vector) $x = (x_1, \ldots, x_n)$ *from the universe* Ω *that minimizes (or maximizes)*

$$f(x) \;=\; f\,(x_1, x_2, \ldots, x_n)$$

In this definition $f(x)$ denotes the objective function; Ω is the feasible region, i.e. it contains all possible decision vectors that can be used to evaluate $f(x)$ and its constraints; $x = (x_1, \ldots, x_n)$ is a decision vector defined in Ω. In this thesis we are interested in findings the global optimum of $f(x)$, hence we will focus on *global search algorithms*. In general, the global optimum of a single-objective optimization problem is defined as follows:

Definition 2. *Let* $f : \Omega \to \mathbb{R}$ *be the objective function to be minimized; let* Ω *be the set of all possible solutions. A solution* $x^* = (x_1^*, \ldots, x_n^*) \in \Omega$ *is a global minimum if and only if*

$$f(x^*) \leq f(x) \quad \forall \;\; x \in \Omega$$

.

For a maximization problem the goal is to find a decision vector $x^* = (x_1^*, \ldots, x_n^*)$, said to be a *global maximum*, such that $f(x^*) \geq f(x)$ for all $x \in \Omega$. Conceptually, the problem consists in finding a solution that provides the minimum (or maximum) value for the objective function within the space of possible solutions.

Many real-world problems require trading off multiple criteria in the same time, including software engineering problems. For example, software re-modularization requires to deal with multiple well-known contrasting criteria, such as *cohesion* and *coupling* [134]. Another example is represented by regression testing that is aimed at selecting test cases that cover (test) as much as possible a program but with minimum execution cost [68]. Differently from single-objective problems, finding optimal solutions for problems with multiple criteria involves to analyze trade-offs. The standard definition of a multi-objectives optimization problem is the following:

Definition 3. *Given a set of objective functions* $F = \{f_1, \ldots, f_m\}$*, where each function* $f_i : \Omega \to \mathbb{R}$*. Let* Ω *be the set of all possible solutions. Find the solutions (or decision vectors)* $x = (x_1, \ldots, x_n)$ *from the universe* Ω *that minimizes (or maximizes) the components of vector*

$$F(x) \;=\; \{f_1(x), \ldots, f_m(x)\}$$

For multi-objectives problems the concept of optimality is based on two widely used notions from economics with wide range of applications in game theory and engineering [68, 135]: *Pareto dominance* and *Pareto optimality* (or *Pareto efficiency*). Without loss of generality, let us assume that we want to minimize a set of objectives $F = \{f_1, \ldots, f_m\}$, the definition of Pareto dominance is the following:

Definition 4. *We said that a solution* x *dominates another solution* y *(also written* $x <_p y$*)*

if and only if the values of the objective functions satisfy:

$$f_i(x) \leq f_i(y) \;\; \forall \, f_i \in F$$
$$\exists \, f_j \in F \;\text{ such that }\; f_j(x) < f_j(y)$$

The definition above means that a solution x *is preferred to* y if and only if, at same level of objective values $f_i \neq f_j$, x is at least better for one objective function $f_j \in F$. Starting from *Definition 4*, the concept of optimality for the multi-objectives problem is defined as follows [135]:

Definition 5. *A solution x^* is Pareto optimal (or Pareto efficient) if and only if it is not dominated by any other solution in the feasible region Ω, i.e., if and only if*

$$f(x^*) <_p f(x) \quad \forall \;\; x \neq x^* \in \Omega$$

The Pareto optimal solution are the ones within the feasible region whose corresponding objective vector components in F cannot be improved simultaneously, hence, no other solution exists which would improve one of the objective functions, without worsening other objectives [135]. While single-objective optimization problems have only one solution, solving a multi-objective problem may lead to find a set of Pareto optimal solutions which, when evaluated, correspond to trade-offs in the objective space. All the solutions that are not dominated by any other decision vectors are said to form a *Pareto-optimal set*, while the corresponding *objective vectors* (containing the values of the objective functions) are said to form a *Pareto frontier*. A decision maker can choose one of the solution within the Pareto frontier according to her preferences or needs.

When the search space becomes too large there is the need for using a specific technique to find efficiently the global optimum (or Pareto optima) for both single-objective and multi-objective problems. This thesis focused on a class of global search algorithms, named *evolutionary algorithms*, inspired on the darwinian principles of *evolution* and *natural selection* [136]. Specifically, this thesis will use single-objective genetic algorithms (GAs) to deal with single-objective problems and multi-objective genetic algorithms for solving multi-objective problems. Next sections describe genetic algorithms and their genetic operators.

2.4 Evolutionary Computation

Charles Darwin theory of *evolution* of species suggests that natural life is an intrinsic search and optimization mechanism. Indeed, biological organisms demonstrate optimized complex behavior where the fittest organisms, i.e. individuals that are more adapted to the natural environment, have higher probability to survive and procreating, then, transmitting their genes to the next generation. The theory of natural selection suggests that organisms that exist nowadays are the result of millions of years of adaptation to the external environment and its resources. The biological organisms showing better characteristics for the natural environment (*fit*) are more capable of earning resources and successfully procreating, hence

they will tends to have numerous descendants. Vice versa, organisms showing worst characteristics (*unfit*) for the natural environment will tend to have few or no descendants. In this way, the characteristics of the fittest organisms are selected over the characteristics of the other organisms by the environment. During reproduction, the recombination of the good characteristics of pairs of fittest organisms can produce more adapted descendants with respect to their parents. Over several generations, organisms evolve to adapt more to the environment.

The efficiency of natural evolution has suggested the possibility to define efficient search algorithms by simulating the behavior of biological organisms. In 1975, Holland developed this idea by abstracting and applying these evolutionary principles into algorithms to solve optimization problems [136]. Evolutionary computation techniques abstract these evolutionary principles into algorithm that can be used to search for optimal solutions to a problem. Genetic Algorithms (GAs) are the most popular and widely applied evolutionary search algorithms [137]. Since their introduction, these algorithms have been widely used in several fields where optimization is required and finding an exact solution is complex, such as industrial engineering, software engineering, numerical analysis, etc. [137].

A GA search starts with a random population of solutions, where each individual (i.e., *chromosome*) of a population represents a potential solution to the optimization problem. Each solution is evaluated on the basis of the function to be optimized to give a measure of its *fitness*. Then, the population is evolved toward better solutions through subsequent iterations to form new individuals by using genetic operators. Specifically, new individuals (i.e., *offsprings*) are generated by applying a *selection operator* that picks out pairs of individuals for reproduction. Since selection operators select the fittest individuals according to their fitness function, it simulates the natural selection due the natural environment. The reproduction (or genetic recombination) is performed through a specific *crossover operator* which creates new individuals (*offsprings*) by combining parts from selected individuals (*parents*). Finally, offsprings are mutated by a *mutation operator*, which randomly modifies their genes, and then evaluated. These genetic operators are performed for several subsequent iterations, called *generations*, until the function cannot be further improved or a fixed amount of time is reached. Hence, generation by generation the population evolves under specific selection rules by adapting itself to the fitness function to be optimized. In general, after some generations the algorithm converges to the best individual, which hopefully represents an optimal or sub-optimal solution to the problem [137]. The advantage of GAs with respect to the other search algorithm is in its intrinsic parallelism because it has multiple solutions (individuals) evolving in parallel to explore different parts of the search space.

In general, a GA has the following basic elements:

- *Encoding* of solutions to the problem;
- *Fitness function* used to measure the quality of each solution; in general, it is proportional to the objective function[2] to be optimized;

[2]Either a vector of objective functions for multi-objective problems

- *Selection operator* to select best individuals within each generation; typically the selection depends on the relative fitness function values of the individuals;
- *Crossover operator*, which is used to recombine pairs of selected individuals;
- *Mutation operator*, which randomly modifies genes of individuals.

In the original definition by Holland [136], a solution is encoded by fixed-length binary string where each element of such a string represents a particular binary decision variable to the problem. In this definition, the genetic operators are binary operators used to deal with binary strings. However, during the years several variants of GAs and genetic operators have been proposed in order to customize the original scheme to different kinds of problems to solve. For example real-coded GAs have been proposed to deal with continuous real problems [137], while permutation operators have been designed to solve permutation problems. All these variants of GAs, as well as the original version by Holland, are often named as *single-objective* GAs because they are designed and developed to solve single-objective problems. More sophisticated GAs have been designed to solve multi-objective problems where selection schemata have to take into account multiple objective functions in order to establish which individuals in each generation are the fittest ones. These GAs are often called as *multi-objective genetic algorithms* or MOGAs.

In this thesis both single-objective and multi-objective genetic algorithms have been considered to deal with various software maintenance and software testing activities. The next two sections provide further details on both single-objective and multi-objective genetic algorithms, and their genetic operators, used in this thesis.

2.5 Single-objective Genetic Algorithms

A single-objective genetic algorithms is a genetic algorithm developed and designed to solve single-objective problems, i.e., problems consisting in finding global optimum to single objective function. The traditional procedure of the single-objective genetic algorithms is shown in Algorithm 1. The algorithm starts with an initial set of random solutions or population, obtained by randomly sampling the search space (line 3 of Algorithm 1). To produce new individuals, GA first creates new offsprings by merging the genes of two individuals in the current generation using a *crossover* operator or modifying a solution using a *mutation* operator (line 5 of Algorithm 1). Then the next generation is obtained by selecting the best (fittest) individuals from parents and offsprings using a *selection* operator and according to their fitness values (line 6 of Algorithm 1). In single-objective optimization, the fitness of an individual is assigned proportionally to the value of the single objective function for that individual. The better the value of the objective function, the higher the corresponding fitness value. The main loop of the algorithm is repeated until a stop condition is reached.

Before applying Algorithm 1 to solve a real-world problem, it is necessary to use an encoding schema to represent solutions as chromosomes. Then, it is needed to choose good genetic operators —that are crossover, mutation, and selection— according to the peculiarity

Algorithm 1: Genetic Algorithms

```
Input:
Number of decision variables N
Population size M
Result: A solution S to the problem
1 begin
2   t ⟵ 0 // current generation
3   generate initial population P_t
4   while not (end condition) do
5     generate offsprings Q_t using crossover and mutation
6     select P_{t+1} from P_t and Q_t
7     t ⟵ t + 1
8   S ⟵ best individual of P_t
```

of the problem taken into account. There are several encoding schemata and genetic operators that can be placed in the pseudo-code of Algorithm 1.

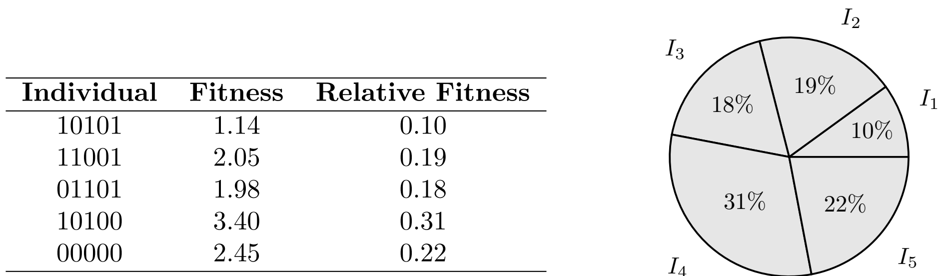
2.5.1 Solution encoding

Encoding a generic solution to a problem into a chromosome is the first issue to consider when using genetic algorithms. Before running GAs it is necessary to represents all possible solutions as chromosomes. Given an optimization problem $min\ f(x_1, x_2, \ldots, x_n)$ where $f(x)$ is the function to be optimized and $x = (x_1, x_2, \ldots, x_n)$ is the vector of decision variables; the simplest encoding schema consists of representing each potential solution by a string s of length n obtained by concatenating all its decision variables:

$$s = x_1 x_2 \ldots x_n$$

The string s is the chromosome that consists of n *genes*, where each gene corresponds to a specific decision variable to the problem; different values of a gene are named *alleles*. The chromosome s is the genotype of the individual while the value assumed by the function f in s is the corresponding phenotype. Hence, the function f links genotype and phenotype. In maximization problem the fitness of a solution s is exactly equals to the objective function $f(x)$, hence, solution having higher phenotype have also higher fitness. Vice versa, in minimization problems, solutions with higher fitness are those showing a lower phenotype.

According to type of alleles of a gene, the encoding schemata can be classified in different categories [138]. In this thesis we consider the two most used encoding schemata that are the *binary encoding* and *real-number encoding*. The *Binary encoding* is the original schema defined by Holland [136] where solutions are represented by binary strings and each gene is a binary decision variable assuming two possible values $\{0; 1\}$. This encoding schema is used for solving binary problems such as the knapsack problem and it will be used in this



Individual	Fitness	Relative Fitness
10101	1.14	0.10
11001	2.05	0.19
01101	1.98	0.18
10100	3.40	0.31
00000	2.45	0.22

Figure 2.1: Example of roulette wheel selection

thesis to solve the test suite optimization problem. The binary encoding schema can be also used for solving non-binary problems by converting integer or real decision variables in binary strings. For example, a vector of three integers <112, 255, 52> in the range $[0, 255]$ can be represented as <01110000, 11111111, 00110100>. For real values, a decision must be made on the precision and the mapping in binary strings to use. To overcome this issue, Antonisse [139] suggested to use a *real number encoding.* In this schema, a chromosome is represented as string of real numbers and each gene is a real decision variable assuming value in real and continuous intervals of the search space. Davis [140] demonstrated that real-valued representations always outperformed binary encodings when solving problems with real-number decision variables. Thus, this schema will be used in this thesis to solve software maintenance activities having real-coded decision variable.

For more complex real-world problems other encoding schemata can be used to capture the properties of the problem itself. A more detailed survey on other encoding schemata can be found in [138].

2.5.2 Selection operators

Selection operators emulate the Darwinian selection due to natural environment. It is used to select fittest individuals that have to survive in the next generation. The intuitive idea is that this operator should be related to the fitness of each individual in some way. The selection operator proposed by Holland is the *roulette wheel selection* [136], which assigns to each individual a probability to survive in proportion to its fitness: the higher the fitness of an individual, the higher its probability to survive or to be selected. The process is analogous to the use of a roulette wheel where each individual share a slice of the wheel that is proportional to its fitness value. Thus, the wheel is spun M times in order to pick M individuals and to make constant the population size. Figure 2.1 shows an example of survival probability assignment obtained using the roulette wheel selection. Despite its simplicity, this operator has several disadvantages. First, it requires that the fitness function has to be positive. Second, in the case of a population having individuals with small fitness values, they have very few chances to be selected if one chromosome has a very high fitness value [137].

To overcome such issues, many selection operators have been proposed in literature [138]. This thesis considers the following two widely used selection operators:

- *Ranking selection.* In this operator, all individuals of the population are ordered according to their fitness values. Then, each individual has the probability to survive that is proportional to its position (rank) in the ordered list, rather than directly using of fitness values. This methodology allows to reduce the effect of individuals having too large fitness values with respect to other individuals. On the other hand, it ignores the magnitude of the fitness differences between individuals.

- *Tournament selection.* This method randomly chooses a set of k individuals from the current population and select the fittest individual within k selected ones. The process is repeated until the maximum number of individuals is selected, i.e. until the population size is reached. The number of individuals k in the set is named *tournament size*. With this operator, individuals with small fitness values have higher chance to be selected with respect to the roulette wheel selection. However, it presents a further controlling parameter[3] that directly affect the selective pressure [137].

Further details of advantages and disadvantages of the two operators can be found in [138].

2.5.3 Crossover operators

Crossover is the operator used to create offsprings by combining genes of their parents. Since this operator works at gene level it is highly dependent of the encoding schema used to represent the solutions. Indeed, each encoding schema has an own plethora of crossover operators. *Single-point crossover* is the simplest and the original crossover designed by Holland for binary encoding and it is outlined in Algorithm 2. It takes two parents and cuts their chromosome strings at some randomly chosen position and the produced substrings are then swapped to produce two new full-length chromosomes. For example, given two binary-coded chromosomes 000000001111111100000000 and 111111110000000011111111, the single-point crossover with cut point in the 12-th position will recombine the two parents generating two new offsprings as follows:

$$\begin{array}{l|l} 000000001111 & 111100000000 \\ 111111110000 & 000011111111 \end{array} \quad \Longrightarrow \quad \begin{array}{l|l} 000000001111 & 000011111111 \\ 111111110000 & 111100000000 \end{array}$$

Crossover is applied to individuals selected at random with a probability p_c, referred to as the *crossover probability*. When crossover is applied, the offsprings are inserted into the new population, while when crossover is not applied the two parents are simply copied into the new population. This simple crossover operator can be applied for real encoding schema as well. For example, given two real-coded chromosomes <0,255,0> and <255,0,255> the recombination with cut point in the second position, will produce two offsprings: <0,0,255> and <255,255,0>

[3]The larger the value of the tournament size, the stronger the selective pressure.

Algorithm 2: Single-point crossover

Input:
Two parents $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_n)$
Result: Two offsprings $C = (c_1, \dots, c_n)$ and $D = (d_1, \dots, d_n)$

1 **begin**
2 randomly choose one crossover point $p \in \{1, \dots, n-1\}$
3 **for** *i=1 to p* **do**
4 $c_i = a_i$
5 $d_i = b_i$
6 **for** *i=p+1 to n* **do**
7 $c_i = b_i$
8 $d_i = a_i$

Multi-point crossover or uniform crossover, is an extension of the single point crossover that uses multiple cut-points instead of a single one. It takes two parents and cuts their chromosome strings at k randomly chosen cut positions and the produced substrings are then swapped over the k point to produce two new full-length chromosomes. In other words it sample uniformly along the full length of a chromosome. Algorithm 3 outlines the pseudo-code of the multi-point crossover. For example, given two binary-coded chromosomes 000000001111111100000000 and 111111110000000011111111, the three-point crossover with cut point in the 5-th and 12-th positions will recombine the two parents generating two new offsprings as follows:

00000	0001111	111100000000	$\Longrightarrow$	00000	1110000	111100000000
11111	1110000	000011111111		11111	0001111	000011111111

As for single-point crossover, the multi-point crossover can be also applied for real encoding schema. For example, given two real-coded chromosomes <0,255,0> and <255,0,255> the recombination with cut points in the first and second positions, will produce two offsprings: <0,0,0> and <255,255,255>.

Arithmetic crossover is a crossover operator defined for real encoding schema. It combines two selected parent chromosomes to produce two new offsprings by linear combination. Graphically, the two generated offsprings lie on the line segment connecting the two parents. Algorithm 4 outlines the pseudo-code of the arithmetic crossover. For example, given two real-coded chromosomes <0,255,0> and <255,0,255> the arithmetic crossover might produce two new offsprings: <101,50,198> and <154,205,57>.

Further crossover operators have been proposed in literature and a detailed analysis of the different crossovers can be found in [141]. In this thesis we used the single-point crossover for test data generation, the multi-point crossover for test suite optimization problem (binary problem), and arithmetic crossover for the other (real-coded) software engineering problems. It is important to highlight that the choice of using these crossover operators in not random,

Algorithm 3: Multi-point crossover

```
   Input:
   Two parents A = (a_1, ..., a_n) and B = (b_1, ..., b_n)
   Result: Two offsprings C = (c_1, ..., c_n) and D = (d_1, ..., d_n)
1  begin
2      randomly choose k crossover points p_1, ..., p_k ∈ {1, ..., n − 1}
3      index ⟵ 0
4      for i=1 to k do
5          for j=index + 1 to p_k do
6              if k is odd then
7                  c_j = a_j
8                  d_j = b_j
9              else
10                 c_j = b_j
11                 d_j = a_j
12         index ⟵ p_k
```

Algorithm 4: Arithmetic crossover

```
   Input:
   Two parents A = (a_1, ..., a_n) and B = (b_1, ..., b_n)
   Result: Two offsprings C = (c_1, ..., c_n) and D = (d_1, ..., d_n)
1  begin
2      Generate a random number p ∈ [0; 1]
3      for i=1 to n do
4          c_i = p · a_i + (1 − p) · b_i
5          d_i = (1 − p) · a_i + p · b_i
```

but it follows trends and choices of previous work in SBSE and SBST [21, 28, 47, 142].

2.5.4 Mutation operators

After crossover, the chromosomes are subjected to mutation in order to prevent the algorithm to be trapped in local minimum. Mutation is considered as a background operator to maintain genetic diversity in the population because it helps to explore of the search space[137]. This is because it injects diversity by randomly modifying some genes in the population.

There are many different mutation operators for the different kinds of encoding schemata. Since we considers only binary and real encoding schemata, in the following we describe only mutation operators for these two schemata:

- *Bit-flip mutation.* It is the simplest mutation for binary encoding schema that consists

in flipping the genes of each individual with small probability p_m, replacing 0 with 1 and vice versa.

- *Uniform mutation.* It is an extension of the bit-flip mutation for real-encoded GAs. While with binary encoding each gene can only assume value in $\{0, 1\}$, with real encoding each gene assumes value in a real and continuous interval of the search space. Hence, the uniform mutation randomly changes the genes of each individual with small probability p_m, replacing an allele (real number) with another allele (real number) of the feasible region. Formally, given a solution $x = (x_1, \ldots, x_n)$ with lower bound $x^{min} = (x_1^{min}, \ldots, x_n^{min})$ and upper bound $x^{max} = (x_1^{max}, \ldots, x_n^{max})$. The uniform mutation will change with small probability p_m a generic gene x_i of x as follows:

$$x_i = x_i^{min} + p \cdot (x_i^{max} - x_i^{min})$$

  where p is a random number generated within the interval $[0; 1]$.

An important parameter for both the two mutation operators is the mutation probability p_m, i.e. the probability by which the genes are mutated. Whit zero probability none individual will be changed, increasing the ability to find nearby better solutions (better exploitation) but also increasing the probability to converge toward some local optimum. If the mutation probability is 100%, all genes of the individuals will be changed reducing the probability to be trapped in local optimum (better exploration) but GA becomes quite similar to random search [137]. For these reasons, generally the mutation probability should be very small in order to reduce the frequency by which the mutation is applied. A widely used mutation probability is $p_m = 1/n$ where n is the length of the chromosome (or equivalently the number of decision variables to a problem). With this probability, on average, only one gene will be changed for each individual.

2.5.5 Elitism

At each generation it is not guaranteed that the fittest individuals will be selected by a selection operator to survive in the next generation. Moreover the mutation operator could randomly modify the fittest individual, reducing the probability of its survival. This means that if the genetic algorithm finds, at some generation, the global optimum, this solution can be thrown away. Thus, for many problems the convergence speed of a genetic algorithm can be improved by guaranteeing the survival of the best or *elite* individual. The mechanism of preserving the best individual (or the best k individuals) is called *elitism* and it is considered as a standard genetic operator [137].

2.5.6 Termination

The main stopping conditions or convergence criteria used to terminate the main loop of GAs are the following:

- *maximum generation*: GA terminates after a given number of maximum generations, independently from the fitness value of the best individual in the last generation and running time.
- *maximum elapsed time*: GA terminates after a given maximum elapsed time, independently from the fitness value of the best individual in the last generation and the reached number of generations.
- *stall generations*: GA terminates whether the objective function of the best individual is not changed for a specific number of generations.

The algorithm will terminate whether at least one these stop conditions is reached.

2.6 Multi-objective Genetic Algorithms

The main goal of MOGAs is to converge to the Pareto optimal front, or equivalently to find the set of Pareto optimal solutions. The main loop of MOGAs is quite similar to the main loop of single-objective GAs shown in Algorithm 1 since they use the same genetic operators. The main differences between them is represented by the mechanism used to select the fittest individuals. With single-objective GAs, selection operator has to deal with only one objective function; hence, the fittest individuals are those having the best objective function value (or phenotype). With multi-objective problems, there are more objective functions to be considered and then the selection operators have to use the concept of Pareto optimality in order to identify the fittest individuals. At each generation, a MOGA provides a set of non-dominated solutions that represents an approximation of the true Pareto front. MOGAs should promote individuals that are non-dominated by any other solution in the current population, thus hopefully guiding the evolution toward the Pareto optimal front.The approximation of the Pareto-optimal solutions involves to deal with two contrasting goals: (i) minimizing the distance to the optimal front and (ii) maximizing the diversity of the generated solutions.

The first MOGA, called Vector Evaluated Genetic Algorithm (VEGA), was proposed by Schaffer as a natural extension of the single-objective GAs [143]. Indeed, the difference between single-objective GA and VEGA regards the selection operator while crossover and mutation remain as usual. VEGA uses sub-population, each one for each objective function; then the roulette wheel selection is performed separately for each sub-population according to corresponding objective function. Later Goldberg [144] suggested to use a Pareto ranking algorithm that orders the population based on Pareto dominance, such that all non-dominated individuals are assigned the same rank (or importance). The idea is that all non-dominated individuals should have the same probability to be selected for the next generation that should be higher than the one corresponding to individuals that are dominated. Hence, the concept of Pareto dominance is used to rank solutions and to apply selection strategies based on non-domination ranks.

Srinivas and Deb [145] proposed a variant to the Pareto ranking algorithm used by Goldberg, called *Non Dominated Sorting Genetic Algorithm* (NSGA), which assigns different levels of Pareto ranking. The non-dominated solutions forms the first rank of Pareto dominance. Then, individuals belonging to the first rank are ignored and another level of non-dominated solutions is computed. The process continues until all the individuals in the current population are classified in one of the ranking values. The selection operator will assign higher survival probability to the individual having a lower (best) Pareto ranking. To maintain diversity, fitness sharing is also used during the selection process. Further MOGAs have been developed by varying the Pareto ranking mechanism proposed by Goldberg, such as Niched Pareto Genetic Algorithm (NPGA) [146], Strength Pareto Evolutionary Algorithm (SPEA), etc. Different Pareto ranking based MOGAs differ on how the ranking algorithm is implemented and which diversity preserving mechanism is used during the selection process.

The work presented in this thesis exploits one of the most popular MOGAs: the elitist *Non-dominated Sorting Genetic Algorithm* (NSGA-II) developed by Deb [107] which is an extension of NSGA [145].

2.6.1 NSGA-II

The elitist Non-dominated Sorting Genetic Algorithm (NSGA-II) [107] is one of the most popular MOGA where the best solutions —i.e., the current non-dominated solutions— are preserved in the population and participate to the reproduction process for the next generation. It uses an elitist selection operator based on a fast non-dominated sorting approach which ranks all individuals in subsequent Pareto fronts. Individuals with best (lower) non-domination ranks are selected applying elitism for the non-dominated solutions, i.e. solutions belonging to the first Pareto rank are preserved in the next generation. The pseudo-code of NSGA-II is outlined in Algorithm 5. It starts with an initial randomly generated population, obtained by randomly sampling the search space (line 3 of Algorithm 5). The population then evolves through a series of *generations* to find nearby better solutions. To produce the next generation, NSGA-II first creates new *offsprings* by merging the genes of pairs of individuals of the current generation using a *crossover* operator or modifying a solution using a *mutation* operator (function MAKE-NEW-POP [107], line 5 of Algorithm 5). A new population is generated using a *selection* operator, to select parents and offsprings according to the values of the objective functions. The process of selection is performed using the *fast non-dominated sorting* algorithm, which lead the selection of solutions with better ranks in the current population. For individuals with the same rank, the selection process is performed by selecting first those stated in non-crowded areas in order to improve diversity: the individuals that are far away from the rest of the population have higher probability to be selected. The *crowding distance* is used in order to make this kind of crowding-aware selection. Specifically, the function FAST-NON-DOMINATED-SORT [107] in line 7 assigns the non-dominated ranks to individuals parents and offsprings. The loop between lines 10 and 14 adds as many individuals as possible to the next generation, according to their non-dominance ranks. If the number of individuals in the next generation is smaller than the population

Algorithm 5: NSGA-II

Input:
Number of decision variables N
Population size M
Result: A set of non-dominated solutions S

1 **begin**
2 $t \longleftarrow 0$ // current generation
3 $P_t \longleftarrow$ RANDOM-POPULATION(N,M)
4 **while** *not (end condition)* **do**
5 $Q_t \longleftarrow$ MAKE-NEW-POP(P_t)
6 $R_t \longleftarrow P_t \bigcup Q_t$
7 $\mathbb{F} \longleftarrow$ FAST-NONDOMINATED-SORT(R_t)
8 $P_{t+1} \longleftarrow \emptyset$
9 $i \longleftarrow 1$
10 **while** $\mid P_{t+1} \mid + \mid \mathbb{F}_i \mid \leqslant M$ **do**
11 CROWDING-DISTANCE-ASSIGNMENT($\mathbb{F}_i$)
12 $P_{t+1} \longleftarrow P_{t+1} \bigcup \mathbb{F}_i$
13 $i \longleftarrow i + 1$
14 Sort($\mathbb{F}_i$) //according to the crowding distance
15 $P_{t+1} \longleftarrow P_{t+1} \bigcup \mathbb{F}_i[1 : (M- \mid P_{t+1} \mid)]$
16 $t \longleftarrow t + 1$
17 $S \longleftarrow P_t$

size M, then further individuals are selected according to the descending order of crowding distance in lines 15–16. The CROWDING-DISTANCE-ASSIGNMENT routine [107] in line 11 assigns a crowding distance to each individual as the sum of the distances between such an individual and all the other individuals having the same Pareto dominance rank. Hence, individuals having higher crowding distance are stated in less densely populated regions of the search space. This mechanism is used in order to avoid the selection of individuals that are too similar to each others.

Chapter 3

Finding optimal IR process using GAs

Contents

3.1 Introduction and Motivation 31

3.2 Background and Related work 33

3.2.1 IR methods 35

3.2.2 IR-based traceability recovery 38

3.2.3 IR-based feature location 39

3.2.4 IR-based bug report duplication 40

3.2.5 IR-based source code labeling 41

3.2.6 Choosing the right IR process 42

3.3 Relationship between Clustering Quality and Performances of IR-based techniques 43

3.4 Finding a (Near) Optimal IR process 47

3.5 Finding a (Near) Optimal LDA Configuration 49

3.6 Empirical Evaluation of LSI-GA 51

3.6.1 Research Questions 51

3.6.2 Scenario I: Traceability Recovery 52

3.6.3 Scenario II: Feature Location 56

3.6.4 Scenario III: Duplicated Bug Report 60

3.7 Empirical Evaluation of LDA-GA 63

3.7.1 Research Questions 63

3.7.2 Scenario I: Traceability Recovery 64

3.7.3 Scenario II: Feature Location 68

3.7.4 Scenario II: Source Code Labeling 72

3.8 Threats to Validity 74
3.9 Conclusion and Future Work 76

3.1 Introduction and Motivation

Researchers indicated that some 40% to 60% of the maintenance effort is devoted to understanding the software to be modified [115]. Indeed, before making changes, the software engineers have to understand the behavior of the software being modified, its main functionalities, its structure and the dependencies between its components (*program comprehension*) [111]. This task becomes much more complex when the individuals that have to make the changes did not develop the software. Program comprehension is a prerequisite of any software maintenance activity because"*software that is not comprehended cannot be changed*" [147]. This process requires to read the source code and its documentation to build a mental representation of the program to be understood and changed [70]. Individuals did not develop the software should understand as quickly as possible where to make a change or a correction in software. However, understanding source code is more difficult than understanding textual document written in natural language because, for example, it is often difficult to trace the evolution of software across different releases/versions if changes are not documented. Thus, program comprehension can be tedious, error prone and time consuming in large software systems, where the developer is requested to read (and comprehend) a large number of source code lines. All these issues has led to a growing body of research works focused on the usage of Information Retrieval (IR) techniques to support the comprehension of source code when addressing software maintenance and development tasks.

All IR-based solutions to SE tasks, such as LSI [78] or LDA [79], require configuring different components, such as type of pre-processors (e.g., splitting compound or expanding abbreviated identifiers; removing stop words and/or comments), stemmers (e.g., Porter or snowball), indexing schemata (e.g., term frequency - inverse document frequency), similarity computation mechanisms (e.g., cosine, dot product, entropy-based), etc. Moreover, other than these commons components, each IR method requires to calibrate a set of its internal parameters. For example, the fast collapsed Gibbs sampling generative model for LDA requires setting four internal parameters: the number of topics k, the number of iterations n, the Dirichlet distribution parameters α and β [148]. Most of existing IR-based approaches to SE tasks rely on ad-hoc methods to configure these solutions, components, and their configurations. A systematic review of SE literature shows that a large number of papers do not even provide the details on how certain IR-based techniques have been instantiated; whereas the remaining set of papers use ad-hoc configurations, component settings, thus, significantly underachieving potential of IR methods to solve SE tasks. In addition, this makes the practical usage of IR-based processes quite difficult and it might hinder the usage of such processes in industrial context. In other cases, the IR process is instantiated using the same parameters and configurations that were originally designed and tested for natural language corpora. This is because the underlying assumption was that source code (or other software artifacts) and natural language documents exhibit similar properties.

Even though IR-based solutions was successfully used in natural language analysis community, applying it on software data, using the same parameter values used for natural language text, did not always produce the expected results [80]. As in the case of machine

learning and optimization techniques, a poor parameter calibration or wrong assumptions about the nature of the data could lead to poor results [81]. Therefore, recent researches have debate about the assumption that source code files are similar or not to documents written in natural language. Hindle *et al.* [82] showed that text extracted from source code is much more repetitive and predictable as compared to natural language text. According to their recent empirical findings, "*corpus-based statistical language models capture a high level of local regularity in software, even more so than in English*" [82]. In our previous work we also demonstrate that during software maintenance and development, the developers typically use a technical language to describe a software system (its functionalities, its architecture and the main dependencies between system components) [17]. Furthermore, documents in traditional textual repository (such as on the web) are heterogeneous, while software documents are much more homogeneous: for example a class is a crisp abstraction of a domain/solution object, and should have few and clear responsibilities.

We conjecture that the (sometime) low performance of IR-based solution can be due to the usage of the same parameters and configurations that were originally designed and tested for natural language corpora. Starting from this conjecture, this chapter presents a search-based approach for calibrating all the steps of an IR-process to achieve better (acceptable) performance on software engineering tasks using unsupervised metrics based the quality of the clustering. The proposed solution, named LSI-GA, takes into account not only task specific components and data sources (i.e., different parts of software artifacts related to solving a particular SE task), but also internal properties of the IR model built from the underlying dataset using a large number of possible components and configurations. We use Genetic Algorithms (GAs) to effectively explore the search space of possible combinations of instances of IR process components (e.g., pre-processors, stemmers, indexing schemata, similarity computation mechanisms) to select the candidates with the best expected performance for a given dataset used for a SE task. Noticeably, using LSI-GA the quality of a solution (represented as a GA individual) is evaluated based on the quality of the clustering of the indexed software artifacts. For this reason, it is unsupervised and task-independent, whereas the instantiated process is dataset-specific. To provide a further evidence to our conjecture, this chapter also uses the same unsupervised search-based approach for automatically calibrating the internal parameters of LDA, which is the most sophisticated IR methods used for SE tasks, with a fixed pre-processing. In the following, this LDA-oriented approach is referred to as LDA-GA to make a distinction between it and LSI-GA.

The contributions of this chapter can be summarized as follows:

- It introduces LSI-GA, an unsupervised search-based approach for automatically assembling IR process on software text corpora using a GA. It also presents an unsupervised search-based approach, named LDA-GA, to automatically calibrate the internal parameters of LDA.

- This chapter shows the generality and applicability of the proposed approaches to different software engineering tasks, named (i) traceability recovery, (ii) feature location, (iii) bug report duplication and (iv) source code labeling.

- An empirical study on LSI-GA demonstrates that IR processes instantiated by LSI-GA outperform previously published results related to the same tasks and the same datasets.
- An empirical study on LDA-GA shows that the LDA configurations obtained by LDA-GA outperform previously reported results and existing heuristics for calibrating LDA.

The chapter is organized as follows. Section 3.2 summarizes background notions and related work on IR methods and their application to SE tasks. Section 3.3 defines the problem of finding a (near) optimal IR process as an optimization problem using the relationship with the quality of clusters. Section 3.4 presents the LSI-GA approach for automatically assembling IR process, while Section 3.5 introduces LDA-GA to automatically calibrate LDA for software engineering tasks. Section 3.6 describes the empirical study and reports the results obtained when applying LSI-GA in the context of (i) traceability recovery, (ii) feature location and (iii) bug duplication. Section 3.7 describes the empirical study and reports the results achieved when applying LDA-GA in the context of (i) traceability recovery, (ii) feature location and (iii) source code labeling. Finally, Section 3.8 discusses the threats to validity that could have affected our study while Section 3.9 reports conclusion and directions for future work.

3.2 Background and Related work

IR methods were proposed and used successfully to extract and analyse textual information in software artifacts focusing on specific software maintenance and development tasks. Specifically, IR methods have been proposed in the literature to recover links between different types of software documents (or artifacts), such as between source code and external documentation [149, 150, 151, 152, 151, 153], among requirements [151], requirements and source code [150, 151, 153], and requirements and test cases [154]. IR methods have been also used for feature location [155], in the context of software measurement to assess the quality of identifiers and comments [156], conceptual cohesion [157] and coupling [158, 159] of classes, as well as assessing and maintaining the quality of external software documentation [160]. In addition, IR techniques have been applied to several other tasks in the past few years, such as bug fix assignment based on problem description reports, identification of duplicate bug reports [161], estimating the time to fix a particular bug based on similar bug reports, classification of software maintenance requests, providing recommendations for novice programmers, identifying developer contributions, mining concept keywords, identification of changes from software repositories, and finding similar software applications.

In general, an IR process follows the steps described in Figure 3.1:

- *Term extraction.* Software artifacts are first indexed in order to identify keywords that characterize the artifact contents. This step also prunes out elements that are not relevant to the IR process, such as white spaces and most non-textual tokens from the text (i.e., operators, special symbols, some numerals, etc.). Then, source code

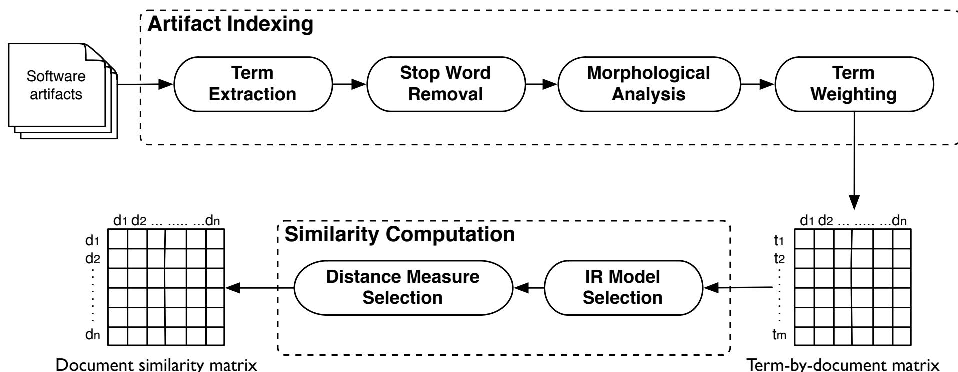


Figure 3.1: Outline of a generic IR Process to solve SE problems.

identifiers composed of two or more words are separated into their constituent words since IR techniques may miss occurrences of concepts if identifiers are not split correctly [162]. To split multi-word identifiers, most existing automatic software analysis tools rely on coding conventions [74] or on more sophisticated natural language processing methods (see e.g., [162, 163, 164, 165, 166]).

- *Stop word removal.* During artifact indexing a stop word function and/or a stop word list are also applied to discard common words (i.e., articles, adverbs, etc) that are not useful to capture the semantics of the artifact content [167, 168]. The stop word function prunes out all the words having a length less than a fixed threshold, while the stop word list is used to cut-off all the words contained in a given word list. Stop word lists are language specific, for example, English has different stop words than Italian. Generally, good results are achieved using both the stop word functions and lists.

- *Morphological analysis.* A more complicated document pre-processing is represented by morphological analysis, like stemming. Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base or root form. There are several existing stemming algorithms, one of the most popular stemmers for the English language is the Porter stemmer [169].

- *Term weighting.* The terms extracted from the documents are stored in a $m \times n$ matrix (called *term-by-document matrix* [167]), where m is the number of all terms that occur within documents, and n is the number of documents in the repository. A generic entry $w_{i,j}$ denotes a measure of the weight (i.e., relevance) of the ith term in the jth document [167]. A widely used weighting schema is the *tf-idf* [167], which gives more importance to words having a high frequency in a document (high *tf*) and appearing in a small number of documents, thus having a high discriminant power (high *idf*). A more sophisticated weighting schema is represented by *tf-entropy* [170], where the local weight is represented by the term frequency scaled by a logarithmic factor, while the

entropy of the term within the document collection is used for the global weight.

- *Application of an algebraic model.* Based on the *term-by-document matrix* representation, different IR methods can be used to measure the textual similarity for each pair of software artifacts, such as VSM [167], LSI [78] and LDA [79].

- *Use of a distance (or similarity) measures.* The last step of the IR process aims at comparing documents, e.g., requirements and source code in traceability recovery, queries and source code in feature location, bug report pairs in duplicate bug report detection. This can be done using different similarity measures. For example, one can use the cosine similarity, the Jaccard similarity, or the Dice (symmetric or asymmetric similarity) coefficient.

An IR process can be instantiated and used in different ways according to a specific software engineering problem taken into account. In general, an IR process is used to compare the textual corpus between different kinds of software artifacts (e.g., for traceability recovery), or to retrieve software documents/atifacts that are related to specific queries (e.g., for feature location or bug report duplication), or just to extract the main semantic concepts from the source code to support the comprehension (e.g., source code labeling).

3.2.1 IR methods

Comprehensive analysis of available research papers reveals that probabilistic models [74, 171], VSM [167, 168, 172], and LSI [78] are the three most frequently used IR methods in software engineering. Only in few cases different methods have been used [173, 174, 16].

Vector Space Model

In the Vector Space Model (VSM), artifacts are represented as vectors of terms (i.e., columns of the term-by-document matrix) that occur within artifacts in a repository [167]. The text similarity is computed using a specific distance function. Typically, the angle between two vectors is used as a measure of divergence between the vectors, and cosine of the angle is used as the numeric similarity (since cosine has the nice property that it is 1.0 for identical vectors and 0.0 for orthogonal vectors). Let $\overrightarrow{D_i}$ and $\overrightarrow{D_j}$ be two documents (or column vectors of the term-by-document matrix), the corresponding textual similarity can be calculated as:

$$cosine(\overrightarrow{D_i}, \overrightarrow{D_j}) = \frac{\overrightarrow{D_i} \cdot \overrightarrow{D_j}}{\|\overrightarrow{D_i}\| \cdot \|\overrightarrow{D_i}\|} \tag{3.1}$$

VSM does not take into account relations between terms. For instance, having "automobile" in one artifacts and "car" in another artifact does not contribute to the similarity measure between these two documents. The VSM has been used to recover traceability links among requirements [175, 151, 176], requirements and source code [174, 74, 176, 177, 153], manual

pages and source code [74, 177, 153], UML diagrams and source code [176, 178], test cases and source code [176], and defect reports and source code [179].

Latent Semantic Indexing

Latent Semantic Indexing (LSI) [78] is an extension of the VSM. It was developed to overcome the synonymy and polysemy problems, which occur with the VSM model [78]. In LSI the dependencies between terms and between artifacts, in addition to the associations between terms and artifacts, are explicitly taken into account. For example, both "car" and "automobile" are likely to co-occur in different artifacts with related terms, such as "motor" and "wheel". LSI assumes that there is some underlying or *latent structure* in word usage that is partially obscured by variability in word choice, and uses statistical techniques to estimate this latent structure. To exploit information about co-occurrences of terms, LSI applies Singular Value Decomposition (SVD) [180] to project the original term-by-document matrix into a reduced space of concepts, and thus limit the noise terms may cause. Basically, given a term-by-document matrix A, it is decomposed into:

$$A = U \cdot \Sigma \cdot V^T \tag{3.2}$$

where U is the term-by-concept matrix, V the document-by-concept matrix, and Σ a diagonal matrix composed of the concept eigenvalues. After reducing the number of concepts to k, the matrix A is approximated with $A_k = U_k \cdot \Sigma_k \cdot V_k^T$. Thus, each term and document is represented by a vector in the k-space of concepts, using elements of V_k^T. Also in this case, the similarity between artifacts is measured as the cosine of the angle between the reduced artifact vectors.

The truncated SVD captures most of the important underlying structure in the association of terms and documents, yet at the same time it removes the noise or variability in word usage that plagues word-based retrieval methods. Intuitively, since the number of dimensions k is much smaller than the number of unique terms m, minor differences in terminology will be ignored. The choice of k is critical: ideally, it is desirable to have a value of k that is large enough to fit all the real structure in the data, but small enough to do not also fit the sampling error or unimportant details. The proper way to make such a choice is an open issue in the factor analysis literature.

LSI has been used to recover traceability links between requirements [151, 176], requirements and source code [174, 176, 152, 177, 153], manual pages and source code [74, 177, 153], UML diagrams and source code [176, 152, 178], test cases and source code [176, 152, 181]. LSI has also been applied for feature location in source code [155], source code labeling [182, 183] and bug report duplication [184].

Latent Dirichlet Allocation

In the probabilistic model, a source artifact is ranked according to the probability of being relevant to a particular target artifact. It represents each document through a probability

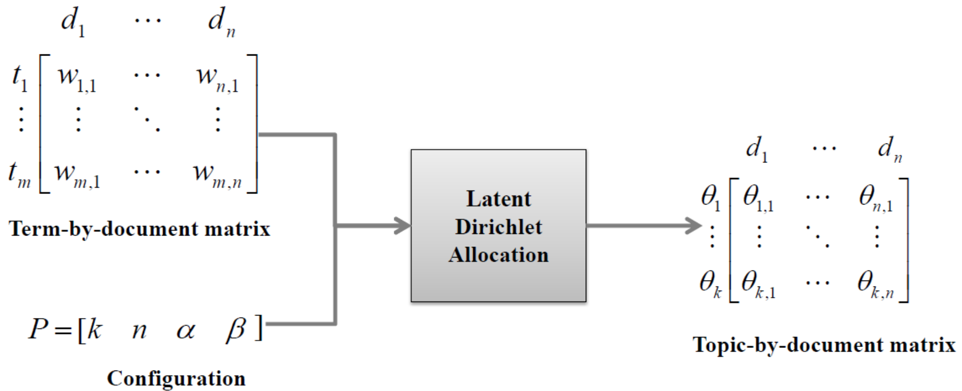


Figure 3.2: LDA process

distribution. This means that an artifact is represented by a random variable where the probability of its states is given by the empirical distribution of the terms occurring in the artifacts (i.e., columns of the term-by-document matrix). The empirical distribution of a term is based on the weight assigned to such a term for a specific artifact. Probabilistic models have been used to recover links between requirements and UML diagrams [171], requirements and source code [185, 186, 74, 187], and manual pages and source code [74].

Latent Dirichlet Allocation (LDA) [79] is a probabilistic model that allows to fit a generative probabilistic model from the term occurrences in a corpus of documents. It takes as input the term-by-document matrix and identifies the latent variables (topics) hidden in the data and generates as output a $k \times n$ matrix θ, called *topic-by-document matrix*, where k is the number of topics and n is the number of documents. A generic entry θ_{ij} of such a matrix denotes the probability of the jth document to belong to the ith topic. Since typically $k << m$, LDA is mapping the documents from the space of terms (m) into a smaller space of topics (k). The latent topics allow us to cluster them on the basis of their shared topics. More specifically, documents having the same relevant topics are grouped in the same cluster, and documents having different topics belong to different clusters.

Figure 3.2 provides a brief overview of how LDA shifts from a term-by-document matrix into a topics space. LDA requires as input a set of hyper-parameters (i.e., a set of parameters that have a smoothing effect on the topic model generated as output). In this work we used the fast collapsed Gibbs sampling generative model for LDA because it provides the same accuracy as the standard LDA implementation, yet it is much faster [148]. For such an implementation, the set of hyper-parameters are:

- k, which is the number of topics that the latent model should extract from the data. To some extent this is equivalent to the number of clusters in a clustering algorithm;
- n, which denotes the number of Gibbs iterations, where a single iteration of the Gibbs sampler consists of sampling a topic for each word;

- α, which influences the topic distributions per document. A high α value results in a better smoothing of the topics for each document (i.e., the topics are more uniformly distributed for each document);
- β, which affects the term's distribution per topic. A high β value results in a more uniform distribution of terms per topic.

Note that k, α, and β are the parameters of any LDA implementation, while n is an additional parameter required by the Gibbs sampling generative model.

3.2.2 IR-based traceability recovery

IR-based traceability recovery aims at identifying candidate traceability links between different artifacts by relying on the artifacts' textual content, that is exactly how IR techniques aim at finding documents relevant to a given query. Traceability recovery works by applying IR techniques to compare a set of source artifacts used as *queries* by the IR method against another set of artifacts considered as *documents* to retrieve. For example, for recovering traceability links between use cases and source code classes, the use cases are used as queries (or *source artifacts*) and the classes are used as *target artifacts.* Both source and target artifacts are typically preprocessed by (i) removing special characters, (ii) splitting identifiers, (iii) removing stop words that are common in the natural language used to write the software documentation either keywords of the program language used when developing the software and (iv) stemming. The standard *tf-idf* is the most used weighting schema for the term-by-document in traceability, while VSM and LSI are among the most used IR techniques. The cosine similarity between all the source artifacts and all the target artifacts is then used to rank all possible pairs of artifacts which represents the potential candidate links. Pairs having a similarity above a certain threshold (fixed by the software engineer), or being in the topmost positions of the ranked list, have higher probability to be candidate links. Therefore, based on a threshold, or based on the number of candidate links a developer could reasonably analyze in the available time, the software engineer cuts and analyzes the top-most part of the ranked list. Based on such an analysis, the software engineer can trace a candidate link (i.e., classify it as true positive), or classifies the link as a false positive. A set of tools that integrates facilities to manage traceability links among different types of software artifacts was developed and evaluated recently [152]. In addition, a recent empirical study highlighted none of these techniques sensibly outperforms the others [80].

Different enhancing strategies—acting at different steps in the process shown in Figure 3.1—have been proposed to improve the performances of traceability recovery methods. In our previous work [17] we observed that the language used in software documents can be classified as a technical language (i.e., jargon), where terms that provide more information about the semantics of a document are the nouns [188]. Thus, we proposed to index the artifacts taking into account only nouns contained in software artifacts. In a further our previous work we proposed the usage of smoothing filters to automatically remove *noise* from the textual corpus of artifacts to be traced [9, 2]. Huang *et al.* [189] and Gibiec *et al.* [190] noted

that one issue which hinders the performances of IR techniques when applied to traceability recovery is the presence of vocabulary mismatch between source and target artifacts. Thus, they propose to use the artifacts to be traced as queries for web search engines and expands the terms in the query artifacts with the terms contained in the retrieved documents before indexing the artifacts. The term weighting could also take into account (i) the structure of the artifacts [171]; and (ii) the importance of a term for a specific domain [191, 192, 175]. As for the latter, artifacts could contain critical terms and phrases that should be weighted more heavily than others, as they can be regarded as more meaningful in identifying traceability links. These terms can be extracted from the project glossary [191, 192] or external dictionaries [175]. Such approaches generally improve the accuracy of an IR-based traceability recovery tool. However, the identification of key phrases (as well as the use of external dictionaries) is much more expensive than the indexing of single keywords.

The term weighting can be changed according to the classification performed by a software engineer during the analysis of candidate links (feedback analysis) [151, 176]. If the software engineer classifies a candidate link as correct link, the words found in the target artifact increase their weights in the source artifact, otherwise, they decrease their weights. The effect of such an alteration of the original source artifact is to "move" it towards relevant artifacts and away from irrelevant artifacts, in the expectation of retrieving more correct links and less false positives in next iterations of the recovery process.

3.2.3 IR-based feature location

The process of applying IR techniques to support feature location is similar to traceability link recovery. In this context IR methods are used to compare a set of short textual descriptions of the bugs or the change requests —used as *queries*— against a set of program elements considered as *documents* to retrieve, such as classes or methods. For example, for locating methods that are related to change requests, the change request are used as queries (or *source artifacts*) and the classes' methods are used as *target artifacts*. The queries can be formulated manually by the developer, or they can be extracted from issue tracking systems —such as Bugzilla— and in this case, the query can consist of the title (e.g., short summary), or the combination of the title and the description of the issue. For the target artifacts, the typical information associated with a method consists of comments, type, name, signature and body and the information associated with a class consists of all the comments, methods and fields. Both queries and target artifacts are typically preprocessed by (i) removing special characters, (ii) splitting identifiers, (iii) removing stop words that are keywords of the program language used when developing the software, (iv) stemming and (v) a weighting schemata (e.g., tf-idf). Also for feature location, LSI is one of the most used techniques [155]. The similarity measure is also in this case the cosine similarity. Using such a similarity measure, the list of target artifacts is ranked descending and presented to the developer which manually investigates these methods and decides if they are relevant or not to the query. A systematic survey of this work related to feature location can be found in [193].

Marcus et al. [155] proposed an Information Retrieval (IR) based approach to concept

location in 2004. This approach also combined with other feature location techniques, such as dynamic analysis [194, 73], program analysis [195], and static analysis [196], cluster analysis [197]. Indeed, many feature location techniques have been combined with IR methods because they allows the user to formulate natural language queries [198]. As for traceability recovery, researchers proposed several enhancing strategies—acting at different steps in the process shown in Figure 3.1– in order to improve the performances of IR-based approaches. Lawrie *at al.* [199] suggested to perform vocabulary normalization by identifiers splitting and identifiers expansion. Dit *at al.* [200] compared human splitting to conservative splitting and found that no significant gains were realized by using perfect/human splitting. Gay *at al.* [198] proposed an approach to augment information retrieval (IR) based concept location via an explicit relevance feedback similarly as proposed in [151, 176] for traceability recovery. Relevance feedback changes the term weighting of queries according to the classification performed by a software engineer during the analysis of the ranked list.

3.2.4 IR-based bug report duplication

For the task of detecting duplicate bug reports, the primary source of information for constructing the corpus consists of the information extracted from issue tracking systems. Each document of the corpus (i.e., each bug) typically consists of the title (e.g., short description of the issue), the description, and in some cases by the project name, component name, severity, priority, etc. In these documents, different weights could be assigned to the previously enumerated elements (e.g., title could be weighted more than description). The *source artifacts* are new, unassigned bugs for which the developer is trying to find similar bugs, and the *target artifacts* are existing bugs which were assigned to developers or resolved. Similarly to the other tasks, the corpus is preprocessed using the standard steps (e.g., removing any sentence punctuation marks, splitting identifiers, removing stop words and stemming). The cosine similarity of an IR technique (e.g., VSM) between the source (i.e., new) bugs and the target (i.e., existing) bugs is used to rank the list of bugs presented to the developer for the manual inspection.

One of the earliest work on finding duplicate bug reports through IR methods is by Runeson *et al.* [76] with the idea that bug reports with high similarity scores are likely to be duplicated. They used a traditional IR process by (i) removing special characters, (ii) removing stop words, (iii) stop word function and (iv) stemming. As weighting scheme they used the simple term frequency *tf*, while as similarity measures the cosine, dice, and jaccard. Wang *et al.* [161] used VSM with *tf-idf* weighting scheme and cosine similarity of the composite feature vectors, then, they selected the top k similar reports as candidate duplicate bug reports. Jalbert and Weimer [201] proposed the same IR process used by Wang *et al.* but using the logarithmic weighting scheme log_{tf} instead of *tf-idf*. More recently, Sureka and Jalote [202] propose an approach that considers n-grams instead of word tokens to characterize bug reports. Sun *et al.* [203] proposed a model based on a two-round stochastic gradient descent to better fit duplicate bug report detection problem.

3.2.5 IR-based source code labeling

For source code labeling, IR techniques have been used to identify keywords that properly describe the artifact in source code identifiers and comments. Such a representation provides a bird-eye's view of the source code artifacts, that allows developers to look over software components quickly, and make more informed decisions on which parts of the source code they need to analyze in detail [204]. Indeed, many researchers have applied IR techniques to automatically "label" software artifacts. For example, Kuhn *et al.* [183] used discriminant words from LSI concepts to label software packages; Thomas *et al.* [205] used LDA to label source code changes; Gethers *et al.* [206] used Relational Topics Model (RTM) to identify and relate topics in high-level artifacts and source code. Maletic and Marcus [182] proposed the combined use of semantic and structural information of programs to support comprehension tasks. Semantic information, captured by LSI, refers to the domain specific issues (both problem and development domains) of a software system, while structural information refers to issues such as the actual syntactic structure of the program along with the control and data flow that it represents. Components within a software system are then clustered together using the combined similarity measure. Baldi *et al.* [207] applied LDA to source code to automatically identify concerns. In particular, they used LDA to identify topics in the source code. Then, they used the entropies of the underlying topic-over-files and files-over-topics distributions to measure software scattering and tangling. Candidate concerns are latent topics with high scattering entropy. Linstead *et al.* [208] used LDA to identify functional components of source code and study their evolution over multiple project versions. The results of a reported case study highlight the effectiveness of probabilistic topic models in automatically summarizing the temporal dynamics of software concerns.

Hindle *et al.* [77] used LDA in an industrial context to relate requirements to code. They performed an empirical study in order to verify whether the information extracted with LDA matches the perception that program managers and developers have about the effort put into addressing certain topics. The results indicated that in general the identified topics made sense to practitioners and matched their perception of what occurred even if in some particular cases practitioners had difficulty interpreting and labeling the extracted topics. Recently, Medini *et al.* [209] used information retrieval methods and formal concept analysis to produce sets of words helping the developer to understand the concept implemented in execution traces. The authors performed both a qualitative as well as a quantitative analysis of the proposed approach. The analysis revealed that the approach is quite accurate in identifying topics in execution traces and in most cases the suggested labeling terms are effective to help grasping the segment functionality. In our previous work [10, 1] we conducted an empirical study involving 38 participants on labeling classes and we pointed out that IR techniques might lead to poor results for some classes and good results for others. This suggests the need of instantiate an appropriate IR process for source code labeling.

3.2.6 Choosing the right IR process

The literature also reports approaches for calibrating specific stages of an IR process, or parameters of specific algebraic IR techniques. Falessi *et al.* [210] empirically evaluated the performance of IR-based duplicate requirement identification on a set of over 983 requirement pairs coming from industrial projects. To this aim, they instantiated 242 IR processes, using various treatments for stemming, term weighting, IR algebraic method, and similarity measure. Their study shows how the performances of the duplicate requirement identification significantly vary for different processes.

Cordy and Grant have proposed heuristics for determining the "optimal" number of LDA topics for a source code corpus of methods, by taking into account the location of these methods in files or folders, as well as the conceptual similarity between methods [211]. Cummins [212] proposed to use genetic programming (GP) to automatically build term weighting formulae, using different combinations of *tf* and *idf*, that can be altered using functions such as logarithm. The similarity between our approach and Cummins' approach is the use of search-based optimization techniques to calibrate IR processes. Their approach was evaluated on a set of 35,000 textual documents, for a document search task. Their approach focuses on term weighting only and it is supervised, as the fitness function evaluation requires the availability of a training set (e.g., labeled traceability links).

Finding an LDA configuration that provides the best performance is not a trivial task. Some heuristics have been proposed [211, 213]; however, these approaches focus only on identifying the number of topics that would result in the best performance of a task, while ignoring all the other parameters that are required to apply LDA in practice. Moreover, such approaches have not been evaluated on real SE applications or have been defined for natural language documents only, thus, they may not be applicable for software corpora. One such technique is based on a heuristic for determining the "optimal" number of LDA topics for a source code corpus of methods by taking into account the location of these methods in files or folders, as well as the conceptual similarity between methods [211]. However, the utility of this heuristic was not evaluated in the context of specific SE tasks. On a more theoretical side, a non-parametric extension of LDA, called Hierarchical Dirichlet Processes [214], tries to infer the optimal number of topics automatically from the input data. Griffiths and Steyvers [213] proposed a method for choosing the best number of topics for LDA among a set of predefined topics. Their approach consists of (i) choosing a set of topics, (ii) computing a posterior distribution over the assignments of words to topics $P(z|w, T)$, (iii) computing the harmonic mean of a set of values from the posterior distribution to estimate the likelihood of a word belonging to a topic (i.e., $P(w|T)$), and (iv) choosing the topic with the maximum likelihood. In their approach, the hyper-parameters α and β are fixed, and only the number of topics is varied, which in practice, is not enough to properly calibrate LDA.

While a number of different SE tasks have been supported using advanced textual retrieval techniques the common problem remains: the way an IR process is instantiated is based on the assumption that the underlying corpus is composed of natural language text. In our survey of the literature, the following SE tasks have been supported using IR-based

approaches and all of these papers and approaches used ad-hoc heuristics to instantiate and IR process, perhaps resulting in sub-optimal performance in virtually all the cases: feature location [215], bug localization [216], impact analysis [217], source code labeling [10], aspect identification [207], expert identification [218], software traceability [173, 219], test case prioritization [220], and evolution analysis [205, 221].

With respect to all the works described above, this Chapter presents and evaluates a search-based approach to instantiate a (near) optimal IR process. It also presents and evaluates a search-based approach to find a (near) optimal LDA configuration. Also, with respect to many other approaches, the approaches discussed in this Chapter *are task independent and does not require any oracle or training set to perform the calibration.* In addition, the outcome of the calibration only depends on the specific artifacts provided as inputs, while it does not depend on the specific task

3.3 Relationship between Clustering Quality and Performances of IR-based techniques

As explained in Section 3.2, the instantiation of an IR process for SE tasks requires to perform several steps, such as term extraction, morphological analysis, term weighting, similarity computation, etc. For each step there is a plethora of possible choices (for example there are several weighting schemata such as *tf* or *tf-idf*), thus, the software engineer can keep one specific sub-process against the others. Therefore, different choices correspond to different instantiations of the IR process which might lead to different performances. Without a guidance that supports the assembly, or with an ad-hoc calibration of the process, the obtained performance may be sub-optimal.

Finding the best configuration of these parameters poses two problems. Firstly, we need a measure that can be used to assess the performances of an IR process before applying it to a specific task (e.g., traceability link recovery). This measure should be independent from the supported SE task. In other words, we cannot simply train an IR process on the data for one particular task, since obtaining such data means solving the task. For example, for traceability link recovery, if we identify all the links to assess the quality of the IR process for extracting the links themselves, then there is no need to have an IR-based model to recover these links anymore. In other words, we need to build such a model on raw data (e.g., source code and documentation) without having additional information about the links. Secondly, we need an efficient way to find the best configuration of parameters, as an exhaustive analysis of all possible combinations is impractical due to (i) the combinatorial nature of the problem (i.e., the large number of possible configuration values for the IR process), as well as (ii) the large amount of computational time required for even such a configuration.

IR methods, such as LSI or LDA, can be considered as a topic-based clustering technique, which can be used to cluster documents in the topics space using the similarities between their topics distributions. For example, LSI is a concept-based clustering technique, which computes the linguistic similarity between source artifacts in the concepts space and clusters

them according to their similarity [222]. This clustering partitions the set of documents into concepts space that represents groups of documents using similar vocabulary. Thus, when using an LSI-based IR process for software engineering tasks, we implicitly cluster SE documents according to the terms/tokens extracted applying a given sequence of pre-processing steps (i.e., stemming, stop word removal, *tf-idf*, etc.) and the number of concepts used for modelling the concepts in space. Such different clusters can be obtained by using (i) various numbers of latent concepts/topics used for modelling the concept/topic space and (ii) various pre-processing techniques.

The conjecture is that there is a strong relationship between the performances obtained by IR processes on software corpora and the produced quality of clusters. Thus, measuring the quality of the produced clusters could provide some insights into the accuracy of IR-based techniques when applied to software engineering tasks. Indeed, if the quality of the clusters produced by an IR-process is poor, this means that the IR process was not able to correctly extract the most important concepts from the software corpus and the documents, which are more similar to each other, are assigned to different clusters (i.e., the IR process assigns different dominant topics to neighbouring documents). We use the concept of a dominant topic to derive the textual clustering generated by a particular IR process applied on a term-by-document matrix. Formally, the concept of a dominant topic can be defined as follows:

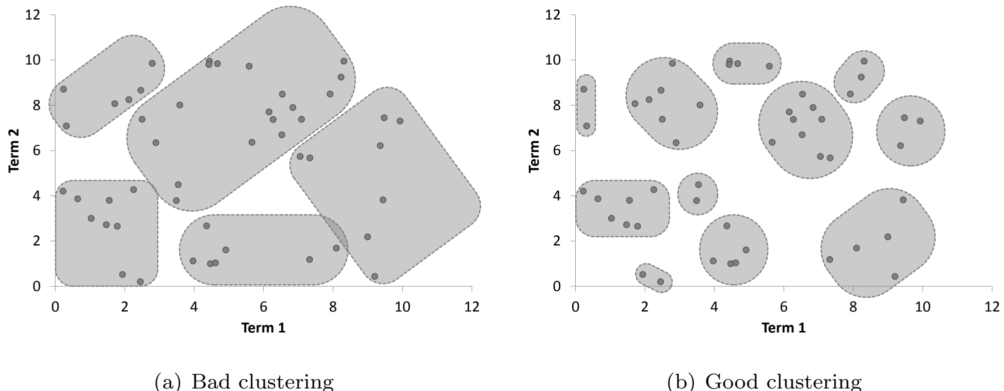
Definition 6. *Let θ be the topic-by-document matrix generated by a particular IR process. A generic document d_j has a dominant topic t_i, if and only if $\theta_{i,j} = max\{\theta_{h,j},\ h = 1 \dots k\}$.*

Starting from the definition of the dominant topic, we can formalize how an IR process clusters documents within the topic space (the number of clusters is equal to the number of topics) as follows:

Definition 7. *Let θ be the topic-by-document matrix generated by a particular instance of an IR process. A generic document d_j belongs to the ith cluster, if and only if t_i is the dominant topic of d_j.*

Thus, we can define a cluster as a set of documents in which each document is closer (i.e., shares the same dominant topic) to every other document in the cluster, and it is further from any other document from the other clusters. It is worth noting that the concept of a dominant topic is specific to software documents only. Collections of natural language documents are usually heterogeneous, meaning that documents can contain information related to multiple topics. In source code artifacts, heterogeneity is not always present, especially when considering single classes. More specifically, a class is a crisp abstraction of a domain/solution object, and should have a few, clear responsibilities. Hence, software documents should be clustered considering only the dominant topic, assuming that each document is related to only one specific topic.

To provide a graphical interpretation of the quality of clusters, Figure 3.3 shows two different clustering models performed on the same set of documents (represented as points in the graphs) in a two-dimensional vector space. The clustering showed in Figure 3.3-a is



(a) Bad clustering

(b) Good clustering

Figure 3.3: Examples of different textual clustering

worst than the clustering shown in Figure 3.3-*b* since documents that are really similar to each other, they are assigned to different clusters. Indeed, by definition an optimal clustering method should group similar documents to form a coherent cluster, while documents that are different should be assigned to different clusters.

Different IR processes (i.e, different configurations) provide different clustering models of the documents. However, not all clustering models that can be obtained by configuring an IR-process are good. There are two basic ways to evaluate the quality of a clustering structure: *internal criteria*, based on similarity/dissimilarity between different clusters and *external criteria*, which use additional and external information (e.g., using judgement provided by users) [223]. Since the internal criterion does not require any manual effort and it is not software engineering task dependent, we use the *internal criteria* for measuring the quality of clusters. More specifically, we use two types of internal quality metrics: *cohesion* (similarity), which determines how closely related the documents in a cluster are, and *separation* (dissimilarity), which determines how distinct (or well-separated) a cluster is from other clusters [223]. There are several measures for cohesion and separation reported in the literature [223] . The simplest way consists of measuring the *cohesion* of a cluster C_i as the mean distance between all the documents in C_i, while the *separation* can be measured as the mean distance between all the documents in C_i and all the documents assigned to the other clusters. Since these two metrics are contrasting each other, we use a popular method for combining both cohesion and separation in only one scalar value, called *Silhouette coefficient* [223]. The Silhouette coefficient is computed for each document using the concept of centroids of clusters. Formally, let C be a cluster; its centroid is equal to the mean vector of all documents belonging to C:

$$Centroid(C) = \sum_{d_i \in C} d_i / |C|$$

Starting from the definition of centroids, the computation of the Silhouette coefficient consists of the following three steps:

1. For each document d_i belonging to the cluster C_k, calculate the *cohesion*, named $a(d_i)$, as the maximum distance between the document d_i and the other documents in the same cluster C_k.
$$a(d_i) = \max_{d_j \in C_k} \{dist(d_i, d_j)\}$$

2. For each document d_i belonging to the cluster C_k, calculate the *separation*, named $b(d_i)$, as the minimum distance from d_i to the centroids of the other clusters, i.e., clusters not containing d_i.
$$b(d_i) = \min_{C_j \neq C_k} \{dist(d_i, C_j)\}$$

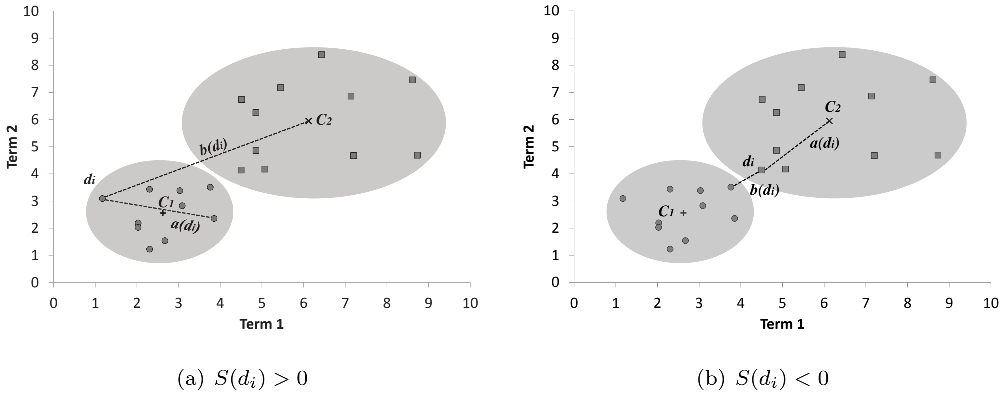
3. For each document d_i, the Silhouette coefficient $s(d_i)$ is then computed as follows:
$$s(d_i) = \frac{b(d_i) - a(d_i)}{\max(a(d_i), b(d_i))}$$

The value of the Silhouette coefficient ranges between -1 and 1. A negative value is undesirable, because it corresponds to the case in which $a(d_i) > b(d_i)$, i.e., the maximum distance to other documents in the cluster is greater then the minimum distance to other documents in other clusters. Vice versa a positive value is highly desirable, because it corresponds to the case in which $b(d_i) > a(d_i)$, i.e., the maximum distance to other documents in the cluster is lower then the minimum distance to other documents in other clusters. For measuring the distance between documents we used the Euclidean distance, since it is one of the most commonly used distance functions for data clustering [223]. Figure 3.4 reports two examples of Silhouette coefficient computed for two different documents: the first graph (a) represents an example of good Silhouette coefficient, since d_i is close to its cluster and very far the nearest document belonging to a different cluster. Instead, the second graph (b) shows a Silhouette coefficient $s(d_i) < 0$, since the distance between d_i and the nearest document belonging to a different cluster is lower than its distance from the centroid of its cluster.

In the end, the overall measure of the quality of a clustering $C = \{C_1, \ldots, C_k\}$ can be obtained by computing the mean Silhouette coefficient of all documents. Let $C = \{C_1, \ldots, C_k\}$ be the clustering obtained using a particular IR process, and let M be an $m \times n$ term-by-document matrix. The mean Silhouette coefficient is computed as:

$$s(C) = \frac{1}{n} \sum_{i=1}^{n} s(d_i)$$

In this thesis, we used the mean Silhouette coefficient as the measure for predicting the accuracy of an IR process in the context of specific software engineering tasks.



(a) $S(d_i) > 0$ (b) $S(d_i) < 0$

Figure 3.4: Example of the Silhouette coefficients.

3.4 Finding a (Near) Optimal IR process

Starting from the conjecture that the higher the clustering quality produced by an IR process, the higher its accuracy when used for software engineering tasks, we can formulate the problem of finding the best IR process as an optimization problem: find the IR process that maximizes the overall quality of the clustering measured using the mean Silhouette coefficient. More precisely, we want to solve the following single-objective problem:

Problem 1. *Finding a set of optimal steps of the IR process* $X = [x_i, x_2, \ldots, n_n]$ *which maximize the average Silhouette coefficient:*

$$\max s(C) = \frac{1}{n} \sum_{i=1}^{n} s(d_i)$$

For solving such an optimization problem we used the single-objective GAs reported in Section 2.5. An IR process consists of a sequence of given steps or components which naturally lends itself to be represented as a chromosome. This chromosome (see Figure 3.5) is a vector, where each *gene* denotes a phase of the IR process, and can assume as a possible value any of the techniques/approaches available for that phase. Each chromosome is composed by six genes $[x_1, x_2, x_3, x_4, x_5, x_6]$, where:

- x_1 is the first gene of the term extraction step which represents the kind of characters to prune out (digit, special characters, etc.);
- x_2 is the second gene of the term extraction step which denotes the technique used for identifier splitting (Camel case splitting, etc.);
- x_3 is the gene for the stop word removal step which represents the techniques applied to discard common words (stop word list either stop word function);

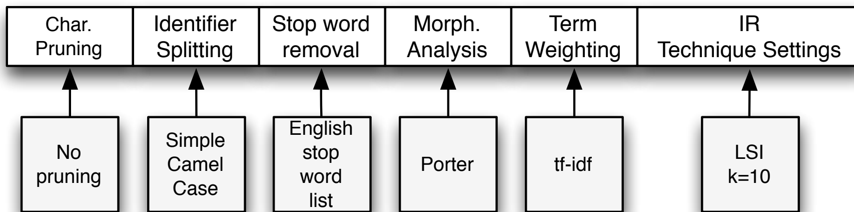


Figure 3.5: LSI-GA chromosome representation.

- x_4 is the gene for the morphological analysis step, which represents a stemming algorithm (Porter stemmer, Snowball stemmer, etc.);
- x_5 is the gene which denotes the weighting scheme (*tf*, *tf-idf*, etc.);
- x_6 it encodes the parameter settings for a given IR method.

While in principle an approach like LSI-GA could also search for the IR method (e.g., LSI, LDA or VSM) achieving the best performances, the fitness function we defined requires at the moment the use of a IR method that clusters documents. For example, LSI [78] and LDA [79] are two such techniques that represent documents using a fixed number of concepts or topics respectively. For this reason, we only considered LSI, which already proves to be very successful for investigated software engineering tasks [219][73]. In terms of calibration, LSI requires to set the number of concepts (k) that is a positive integer value representing the number of topics that the latent model should extract from the data. Thus, an individual (or chromosome) is a particular IR configuration and the population is represented by a set of different IR configurations.

The GA initial population is randomly generated, i.e., by randomly choosing the value of each gene of each individual. The selection operator is the *Roulette wheel selection*, which assigns to the individuals with higher fitness a higher chances to be selected. The crossover operator is the *single-point crossover*, which, given two individuals (parents) p_1 and p_2, randomly selects a position in the chromosome, and then creates two new individuals (the offspring) o_1 composed of the left-side of p_1, and the right-side of p_2, and o_2 composed of the left-side of p_2 and the right-side of p_1. The mutation operator is the *uniform mutation*, which randomly changes one of the genes (i.e., one of the six IR parameter values) of an individual, with a different parameter value within a specified range. The GA terminates after a fixed number of generations or when the fitness function cannot be improved further (i.e., GA converged). Our GA approach can be briefly summarized as (i) generating IR configurations, (ii) using them to cluster documents, (iii) evaluating the cluster quality using the Silhouette coefficient, and (iv) using that value to drive the GA evolution.

LSI-GA has been implemented in *R* [224] using the `GA` library. Every time an individual needs to be evaluated, we process documents using features available in the `lsa` package which allows applying all the pre-processing steps, while for computing the SVD decomposition we

Table 3.1: Values of the genes (steps of the IR process) for LSI-GA.

Step	Implementations
Character pruning	keep special characters and digits remove special characters, keep digits remove both special characters and digits
Identifier splitting	do not split simple Camel Case split Camel Case keep compound split
Stop word removal	do not remove stop words English/Italian stop word removal + remove ≤ 2 char words English/Italian stop word removal + remove ≤ 3 char words
Morphological analysis	no stemming Porter stemming Snowball stemming
Term weighting	Boolean *tf* *tf-idf* $log_{ij} = \log\left(tf_{ij} + 1\right)$ *tf*-entropy
Algebraic model	LSI
LSI k	$10 \leq k \leq rank(TDM)$

used the a fast procedure provided by the `irlba` package for large and sparse matrices. As for the GA settings, we use a crossover probability of 0.8, a uniform mutation with probability of $1/n$, where n is the chromosome size ($n = 6$ in our case). We set the population size equals to 50 individuals with elitism of two individuals. As stop condition for GA, we terminate the evolution if the best fitness function value does not improve for 10 consecutive generations or when reaching the maximum number of generations equals to 100 (which, however, was never reached in our experiments). All such settings are commonly used in the genetic algorithm community. In addition, to address the intrinsic randomness of GAs, for each task and for each dataset we perform 30 independent runs, storing the best configuration and the relative best fitness function value—i.e., the Silhouette coefficient—for each run. Finally, among the obtained configurations, we consider the one that achieves the median fitness function—across the 30 independent runs—for the best individual in the last generation. Finally, Table 3.1 summarizes the possible values for each gene of the chromosome used in our experimentation. Clearly, the number of possible values can easily be extended (e.g. different stemming, different weighting schemas, etc.).

3.5 Finding a (Near) Optimal LDA Configuration

As explained in Section 3.2.1, LDA—and in particular its implementation based on fast collapsed Gibbs sampling generative model—requires the calibration of four parameters, k, n, α, and β. Without a proper calibration, or with an ad-hoc calibration of these parameters,

LDA's performance may be sub-optimal. Starting from the conjecture that the higher the clustering quality produced by LDA, the higher the accuracy of LDA when used for software engineering tasks, we can formulate the problem of finding the best LDA configuration as an optimization problem: find the LDA's configuration that maximizes the overall quality of the clustering produced by LDA measured using the mean Silhouette coefficient. More precisely, we want to solve the following single-objective problem:

Problem 2. *Finding a set of optimal LDA's parameters* $X = [k, n, \alpha, \beta]$ *which maximize the average Silhouette coefficient:*

$$\max s(C) = \frac{1}{n} \sum_{i=1}^{n} s(d_i)$$

For solving such an optimization problem we used the single-objective GAs reported in Section 2.5. Individuals (solutions) are represented as arrays (chromosomes) with four real numbers[1] $[k, n, \alpha, \beta]$, where:

- k is a positive integer value which represents the number of topics that the latent model should extract from the data. We set its upper bound equal to the total number of documents in the dataset.
- n is a positive integer value which denotes the number of Gibbs iterations, where a single iteration of the Gibbs sampler consists of sampling a topic for each word. We set the maximum number of iterations equal to 400.
- α and β are two real hyperparameters which take values within the interval $[0; 1]$.

Thus, an individual (or chromosome) is a particular LDA configuration and the population is represented by a set of different LDA configurations. The selection operator is the *Roulette wheel selection*, the crossover operator is the *arithmetic crossover*, while the mutation operator is the *uniform mutation*. The proposed GA approach can be briefly summarized as follows: (i) generating LDA configurations, (ii) using them to cluster documents, (iii) evaluating the cluster quality using the Silhouette coefficient, and (iv) using that value to drive the GA evolution.

The LDA-GA has been implemented in `R` [224] using the `topicmodels` and `GA` libraries. The former library provides a set of routines for computing the fast collapsed Gibbs sampling generative model for LDA, while the latter is a collection of general purpose functions that provides a flexible set of tools for applying a wide range of GA methods. For GA, we used the following settings: a crossover probability of 0.8, a mutation probability of 0.1, a population of 100 individuals, and an elitism of 2 individuals. As a stopping criterion for the GA, we terminated the evolution if the best results achieved did not improve for 10 generations; otherwise we stopped after 100 generations. All the settings have been calibrated using a trial-and-error procedure, and some of them (i.e., elitism size, crossover

[1] In other words we used the real encoding scheme

and mutation probabilities) were the values commonly used in the community. To account for GA's randomness, for each experiment we performed 30 GA runs, and then we took the configuration achieving the median final value of the fitness function, i.e., the median Silhouette coefficient $s(C)$.

3.6 Empirical Evaluation of LSI-GA

This section describes in details the design and the results of the empirical study we conducted to evaluate the proposed LSI-GA approach. The study was conducted following the Goal-Question-Metric paradigm by Basili *et al.*[225].

3.6.1 Research Questions

The *goal* of our study is to investigate whether LSI-GA allows to instantiate IR processes that are able to effectively solve SE tasks, while the *quality focus* is represented by the performances of the IR-based processes in terms of accuracy and completeness. The *perspective* is of researchers interested in developing an automatic approach to assemble IR processes for solving specific SE tasks. The *context* of the study consists of three SE tasks, namely traceability links recovery, feature location, and identification of duplicate bug reports. Specifically, the study aims at addressing the following research questions (RQs) that have been addressed in the context of the three different SE tasks considered in our study:

- $\mathbf{RQ}_1$: *How do the processes instantiated by LSI-GA compare with those previously used in literature for the same tasks?* This RQ aims at justifying the need for an automatic approach that calibrates IR processes for SE tasks. Specifically, we analyzed to what extent the process instantiated by LSI-GA for solving a specific task is able to provide better performances than a process with an *ad-hoc* setting. Our conjecture is that, with a proper setting, the performances could be sensibly improved because in many cases, the IR-based techniques have been severely under-utilized in the past.

- $\mathbf{RQ}_2$: *How do the processes instantiated by LSI-GA compare with an ideal configuration?* We empirically identified the configuration that provided the best results as compared to a specific oracle. For instance, in the case of traceability recovery, we identified the configuration that provided the best performances in terms of correct and incorrect links recovered. Clearly, one can build such a configuration only with the availability of a labeled data set, by using a combinatorial search among different treatments and by evaluating each combination against the oracle in terms of precision and recall. We call such a configuration *ideal*, because it is not possible to build a priori (i.e., without the availability of a labeled training set) a configuration providing better performances than that. The performances achieved by the process instantiated by LSI-GA are then compared with those achieved with the ideal configuration, to investigate how far off is the LSI-GA configuration from the best possible performances that one can achieve.

Table 3.2: Characteristics of the systems used for traceability recovery.

System	Description	Artifacts		
		Type	Number	Total
EasyClinic	A software system used to manage a doctor's office developed by students	Use Cases	30	77
		Code Classes	47	
eTour	An electronic tourist guide developed by students	Use Cases	58	174
		Code Classes	116	
iTrust	A medical application	Code Classes	33	149
		Java Server Page	116	

To answer to the research questions above we considered three different scenarios named traceability recovery, feature location and bug report duplication. Next section describes the plan of the empirical study and the corresponding empirical results.

3.6.2 Scenario I: Traceability Recovery

The context of this scenario consists of software artifacts extracted from three projects, namely: EasyClinic, eTour and iTrust. The first two systems were developed by the final year Master's students at the University of Salerno (Italy). The documentation, source code identifiers, and comments for both systems are written in Italian. The last system is a medical application used as a class project for Software Engineering courses at the North Carolina State University[2]. All artifacts consisting of use cases and Java Server Pages are written in English. Table 3.2 summarizes the characteristics of the considered software systems in terms of type, number of source and target artifacts. Other than the listed artifacts, each repository also contains the traceability matrix built and validated by the application developers. For different kinds of artifacts, the traceability matrices were developed at different stages of the development (e.g., requirement-to-code matrices were produced during the coding phase). We consider such a matrix as the *oracle* to evaluate the accuracy of the different IR configurations.

To answer to our research questions, we used an IR process with a specific input configuration to recover traceability links between artifact pairs on the term-by-document matrices. In all cases the term-by-document matrix is extracted using all the steps of the process derived by encoded solutions obtained by GAs. To answer both the research questions we performed three different traceability recovery activities:

- A_1: recovering traceability links between use cases and source code classes for EasyClinic. The total number of correct links is 83 while the number of all possible links is 1,410.
- A_2: recovering traceability links between use cases and source code for eTour. The total number of correct links is 246 while the number of all possible links is 6,728.

[2]http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=tracing

- A_3: recovering traceability links between code classes and java server page for iTrust. The total number of correct links is 58 while the number of all possible links is 3,828.

To answer **RQ1**, we compared the accuracy of recovering traceability links achieved by the IR process assembled by LSI-GA with the accuracy achieved by LSI on the same systems in the previously published studies where an “ad-hoc” corpus pre-processing and LSI configuration were used [2, 152]. We also compared the accuracy of recovering traceability links using different combinations of pre-processing steps[3] ($3 \cdot 3 \cdot 3 \cdot 3 \cdot 5$=405) for a different number of concepts for LSI. Specifically, we varied (using step 1) the number of concepts from 10 to maximum number of topics, which is 77 for EasyClinic, 176 for eTour and 80 for iTrust. We also exercised all possible combinations of pre-processing steps with such values. Thus, the total number of trials performed on EasyClinic, eTour and iTrust were 27,135 (it corresponds to the number of all possible preprocessing steps 405 multiplied by the number of all possible number of topics 67), 67,230 and 32,400, respectively. With such an analysis we are able to identify the configuration which provides the best recovery accuracy (as compared with our oracle) between all the possible configurations aiming at estimating the ideal configuration of the IR-based traceability recovery process. We then compared the performances achieved with such an ideal configuration with the performances achieved with the configuration identified by LSI-GA in order to answer **RQ2**.

Metrics

For both $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, the performances of the LSI-GA approach, of the baseline approach, and of the ideal approach are evaluated and compared by using two well-known metrics in the IR field, namely precision and recall [167]. Recall measures the percentage of links correctly retrieved, while precision measures the percentage of links retrieved that are correctly identified:

$$recall = \frac{\mid correct \cap retrieved \mid}{\mid tot.\ correct \mid}$$

$$precision = \frac{\mid correct \cap retrieved \mid}{\mid retrieved \mid}$$

where *correct* and *retrieved* represent the set of correct links and the set of links retrieved respectively. A common way to evaluate the performance of retrieval methods consists of comparing the precision values obtained at different recall levels. This result is a set of recall/precision points which are displayed in precision/recall graphs. In order to provide a single value that summarizes the performance, we use the *average precision*, that can be defined as the mean of the precision scores obtained for each correct link [167]. It can be

[3]For further details see Table 3.1.

mathematically expressed as

$$AP = \frac{\sum_{i=1}^{n} x_i \; prec_i}{\sum_{i=1}^{n} x_i}$$

where x_i represents the binary correctness of ith link (i.e. $x_i = 1$ if the ith is correct; $x_i = 0$ otherwise) while $prec_i$ denotes its precision value.

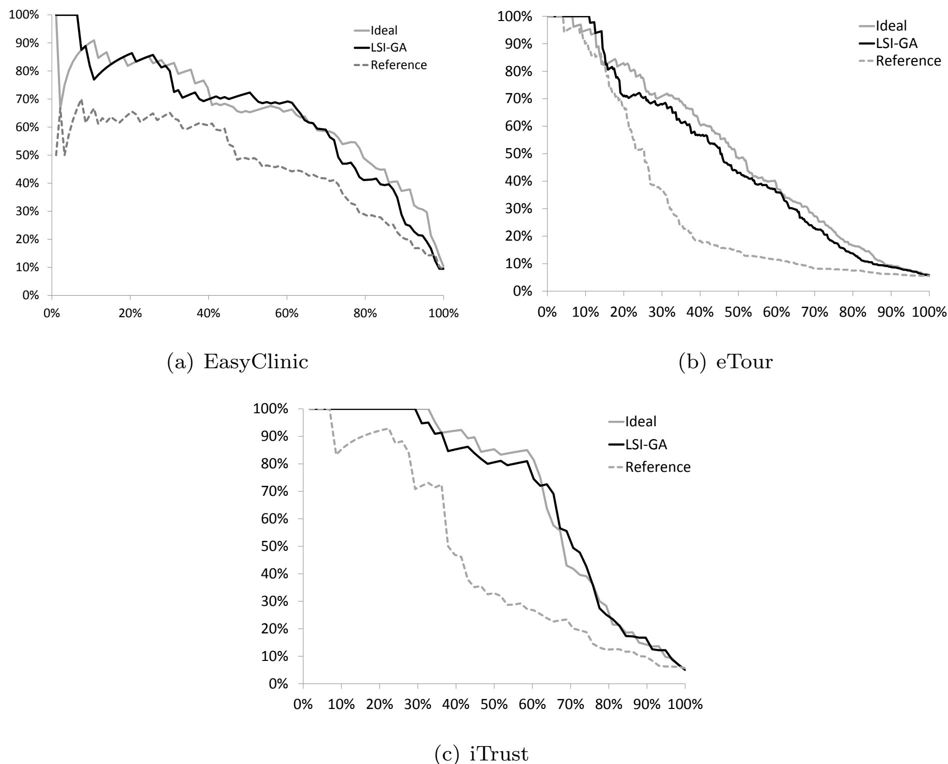
To provide statistical support to our research questions we used a statistical test to verify whether the number of false positives retrieved by one method is statistically lower than the number of false positives retrieved by another method. In other words, we compared the false positives retrieved by method m_i (e.g., LSI with a specific configuration) with the false positives retrieved by method m_j (e.g., LSI with another specific configuration) to test the following null hypothesis:

> *There is no statistically significant difference between the number of false positives retrieved by m_i and m_j*

The dependent variable of our study is represented by the number of false positives retrieved by the traceability recovery method for each correct link identified. Since the number of correct links is the same for each traceability recovery activity (i.e., the data were paired), we used the Wilcoxon Rank Sum test [226]. In all our statistical tests we consider p-values $<$ 0.05 as statistically significant. Since this requires performing three tests for each system, we adjusted the p-values using Holm's correction procedure [227].

Empirical results

Figure 3.6 reports the precision/recall graphs obtained by LSI using (i) the *ideal* IR configuration obtained in combinatorial search across 27,135 (EasyClinic), 67,230 (eTour) and 32,400 (iTrust) different configurations; (ii) the IR configuration identified by LSI-GA; and (iii) an "ad-hoc" configuration (i.e., *reference*) used in a previous study, where LSI was used on the same dataset and for the same traceability recovery task. For all three systems, LSI-GA is able to obtain a precision/recall curves close to the ones yielded by the ideal configurations. It is also important to highlight the ideal configurations are obtained using the oracles, thus, they are used just to show the upper (ideal) bounds. If comparing the performance achieved by LSI-GA with those of the reference configuration, we can observe a tangible improvement in all cases. For example, on EasyClinic for a recall level smaller than 80%, using LSI-GA it is possible to achieve an improvement of precision ranging between 50% and 15% with respect to the reference configuration [2, 152]. A less evident improvement is achieved for the lowest recall percentile (i.e., between 80% and 100%). A similar analysis can be performed for the other two software systems where the improvements are even better: for example on iTrust at recall level of 60% LSI-GA reaches 80% of precision against the 30% of the reference IR configuration obtained at same level of recall.



(a) EasyClinic

(b) eTour

(c) iTrust

Figure 3.6: Traceability recovery: precision/recall graphs.

This scenario is also confirmed by the average precision obtained by the three different treatments (Table 3.3). Indeed, the average precision obtained by LSI-GA is quite close to the ideal one. In other words, the difference in terms of average precision with respect to the ideal configuration is always lower than 3%. However, the improvement obtained with respect to the reference configuration is of about 20% in terms of average precision. These results are also confirmed by our statistical analysis. Table 3.12 reports the results of the Wilcoxon test (i.e., the adjusted p-values) for all combinations of techniques; statistically significant results are highlighted in bold face. As we can see, there is always statistical significant difference between the performance obtained LSI-GA and he reference configurations. However, for all the systems the ideal configurations (global optima) are statistically better than the LSI-GA. Anyhow the precision/recall graph reported in Figure 3.6 reveal the difference is relatively small especially if compared to the improvement earned with respect to the reference approaches.

Table 3.5 reports the different IR pre-processing steps and the k values for LSI found by the LSI-GA, and compares them with the ones generated by the ideal IR process and the ones used by the baseline. As we can see in the majority of cases the pre-processing steps

Table 3.3: Comparison of traceability recovery performances: average precision values.

System	**Ideal**	**LSI-GA**	**Reference** [2, 152]
EasyClinic	64.79	64.68	46.78
eTour	50.18	47.32	30.93
iTrust	68.56	67.94	45.47

Table 3.4: Comparison of traceability recovery performances (precision): results of the Wilcoxon test.

	EasyClinic	**eTour**	**iTrust**
LSI-GA < Ideal	1	1	0.97
LSI-GA < Reference [2, 152]	< **0.01**	< **0.01**	<**0.01**
Ideal < Reference [2, 152]	< **0.01**	< **0.01**	< **0.01**

chosen by LSI-GA are the same for the ideal/optimal IR with only few differences. If we compare the reference IR processes with those instantiated by LSI-GA, one can notice that LSI-GA often chooses a less aggressive term extraction (e.g., keeping digits) and stop word removal (e.g., not filtering out short words). For the choice of the number of concepts/topics to be used when applying LSI we observe that the k values provided by LSI-GA are quite close to those utilized by the ideal IR process. Hence we can conclude that LSI-GA allows us to instantiate an IR process that is close to the ideal one. It is also interesting to note that not all the dataset require the same pre-processing steps (e.g., stopword list, stopword function and stemming), confirming the findings of Falessi *et al.* [210] that there is no unique IR process that can be efficiently applied to all the tasks and all the datasets.

3.6.3 Scenario II: Feature Location

In this scenario we consider two Java software systems, namely *jEdit v4.3* that is an open-source text editor for programmers, and *JabRef v2.6* that is a bibliography reference manager. Table 3.13 reports the system size (in KLOC), the number of source files and methods, and the number of bugs and features to be located. These systems have been used in previous studies on feature location [215, 228]. For these systems are available (i) the gold sets (i.e., mappings between source code and features) at method level granularity, (ii) the description of the features from bug reports and (iii) execution traces.

To answer to our research questions, we used LSI with a specific input parameters to extract topics from the source code and measuring the textual similarity between methods and description of feature. Then, the list of methods are ranked by their similarity to the description of the feature. In all cases the term-by-document matrix is extracted using all the steps of the process derived by encoded solutions obtained by GAs. For **RQ1**, we compared the performances obtained by the IR process assembled by LSI-GA with the performances achieved by LSI on the same systems in a previously published study, where an “ad-hoc” reference IR pre-processing and LSI were used [195]. The latter is used as a baseline. To

Table 3.5: Comparison of different IR processes provided by LSI-GA, ideal and reference.

System	Method	k	Special Chars	Digit Chars	Term Splitting	Stopword List	Stopword Functions	Stemming	Weight Schema
EasyClinic	Ideal	72	Remove	Include	Camel Case	No	No	Porter	tf-idf
	LSI-GA	66	Remove	Include	Camel Case	Yes	No	No	tf-idf
	Refer.	37	Remove	Remove	Camel Case	Yes	Yes	Snowball	tf-idf
eTour	Ideal	160	Remove	Remove	Camel Case	No	No	Snowball	tf-idf
	LSI-GA	163	Remove	Include	No	No	Yes	Snowball	tf-idf
	Refer.	87	Remove	Remove	Camel Case	Yes	Yes	Snowball	tf-idf
iTrust	Ideal	80	Remove	Remove	No	Yes	No	Porter	tf-idf
	LSI-GA	69	Remove	Include	Camel Case	Yes	No	Porter	tf-idf
	Refer.	40	Remove	Remove	Camel Case	Yes	Yes	Snowball	tf-idf

Table 3.6: Task 2 (feature location): characteristics of the datasets used in the experiment.

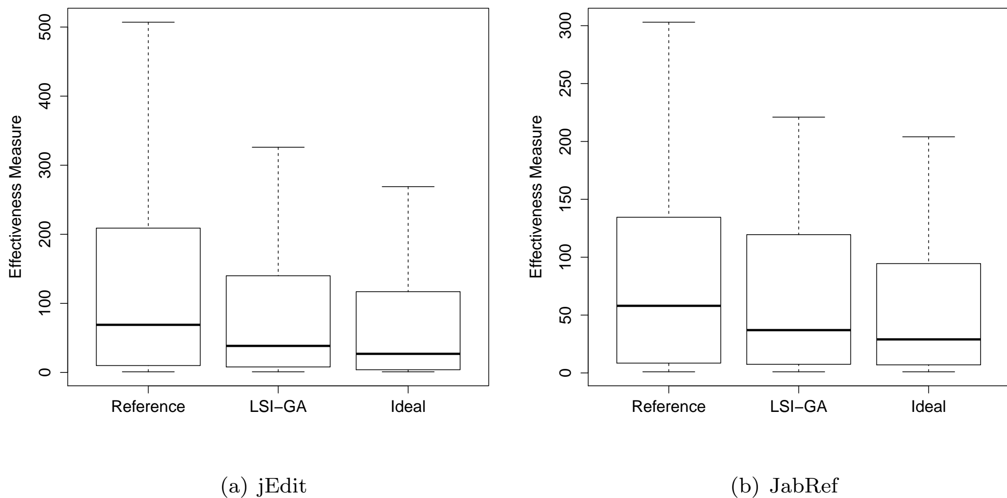
System	Version	KLOC	Files	Methods	Features
jEdit	4.3	104	503	6,413	150
JabRef	2.6	74	579	4,607	39

address **RQ2**, we compare the process instantiated by LSI-GA with an "ideal" process, which was determined by using a combinatorial approach similar as done for traceability recovery. Specifically, we varied the number of topics from 100 to 1,500 with step 10 for both jEdit and JabRef. We also applied all possible combinations of preprocessing steps considered in this work. Thus, the total number of trials performed on both software systems consisted is of 60,912. With such an analysis we are able to identify the configuration which provides the best accuracy (as compared with our oracle) between all the possible configurations aiming at estimating the ideal configuration of the IR-based traceability recovery process.

Metrics

In this scenario the performances of the IR processes were analyzed using the *effectiveness measure (EM)* [73]. The EM estimates the number of methods a developer needs to analyze before finding the first relevant method, and it is computed as the lowest rank of a relevant method found in the list of methods sorted in a descendent order by their textual similarity to the description of the feature of interest. A high EM value suggests bad performance of the IR technique and a greater human effort, due to the large number of false positive methods to be analyzed before finding a relevant one. In order to provide a single value that summarizes the performance, we use the *average effectiveness measure*, that is equal to the mean of the EM scores obtained for each feature to locate [73].

To provide statistical support to our research questions we used a statistical test to verify whether the EM values obtained by one method m_i (e.g., LSI with a specific configuration) is statistically significantly lower than the EM values yielded by another method m_j (e.g., LSI with another specific configuration) similarly as done in previous work [73, 193]. In other words, we want to test the following null hypothesis:



(a) jEdit

(b) JabRef

Figure 3.7: Box plots of the effectiveness measure for feature location on jEdit and JabRef.

> *There is no statistically significant difference between the EM values obtained by* m_i *and* m_j

The dependent variable of our study is represented by the EM values reached by the feature location methods for each feature to be located. Since the number of features to be located is the same for each instantiated IR process (i.e., the data were paired), we used the Wilcoxon Rank Sum test [226]. In all our statistical tests we consider p-values < 0.05 as statistically significant. Finally, the p-values are adjusted using Holm's procedure.

Empirical results

Figure 3.7-a shows boxplots of the EM values for the jEdit system, that were computed from three different IR process instances: (i) the ideal IR process obtained over 60,912 possible configurations; (ii) the IR process assembled by LSI-GA; and (iii) the "ad-hoc" IR process (i.e., the reference) used in a previous study [195]. First, we can observe that the EM values obtained by LSI-GA are quite close to the ones obtained by the ideal IR process, thus, LSI-GA was able to find a near optimal configuration. When comparing the performance of LSI-GA with the reference, we observe that there is a substantial difference for all the quartiles. For example, the median and mean EM values achieved by LSI-GA are 38 and 197, whereas the median and mean EM values achieved by the reference are 69 and 244, respectively. Similar results are observed for the boxplots for JabRef shown in Figure 3.7-b. The EM distribution provided by LSI-GA is comparable to the distribution obtained by the ideal IR process. Moreover, LSI-GA provides EM values that are better than those provided

Table 3.7: Comparison of feature location performances (EM): the results of the Wilcoxon test.

	jEdit	**JabRef**
LSI-GA $<$ Ideal	1	0.27
LSI-GA $<$ Reference [195]	$<$ **0.01**	0.40
Ideal $<$ Reference [195]	$<$ **0.01**	0.20

Table 3.8: Comparison of different IR processes provided by LSI-GA, ideal and reference.

System	**Method**	**k**	**Special Chars**	**Digit Chars**	**Term Splitting**	**Stopword List**	**Stopword Functions**	**Stemming**	**Weight Schema**
jEdit	Ideal	1150	Remove	Remove	Camel Case Keep-Comp.	Yes	$<$ 3 char.	Snowball	tf-idf
	LSI-GA	911	Remove	Remove	Camel Case Keep Comp.	Yes	$<$ 3 char.	Snowball	tf-idf
	Reference	300	Remove	Remove	Camel Case	Yes	Yes	Porter	tf-idf
JabRef	Ideal	416	Remove	Remove	Camel Case	Yes	$<$ 3 char.	Snowball	tf-idf
	LSI-GA	390	Remove	Remove	Camel Case	Yes	$<$ 3 char.	Snowball	tf-idf
	Reference	300	Remove	Remove	Camel Case	Yes	Yes	Porter	tf-idf

by the reference IR preprocessing. Indeed, the median EM value produced by LSI-GA is 36, while the median EM values produced by the ideal and the reference are 30 and 58, respectively.

To provide a statistical support to our preliminary results, the Wilcoxon Rank Sum test was used to test if the difference between the effectiveness measures of two feature location techniques is statistically significant or no. Table 3.7 reports the obtained adjusted p-values for all combinations of the techniques. On jEdit, LSI-GA achieves significantly better results than the reference process, while the ideal IR process performs significantly better than LSI-GA. However, from the boxplots of Figure 3.7, we can observe that the difference between LSI-GA and the ideal process is relatively small. Turning to JabRef, even if the boxplots revealed a better EM distribution for both LSI-GA and ideal as compared to reference, the results of the Wilcoxon tests indicate that these differences are not statistically significant. This suggests that the "ad-hoc" configuration identified by Liu *et al.* [195] on the JabRef is the one providing results very close to the best. However, the same "approach" did not provide the same level of accuracy on jEdit, where the "ad-hoc" configuration is not able to rich the same level of accuracy of both the ideal and LSI-GA configurations.

Table 3.5 reports the different IR pre-processing steps and the k values for LSI found by the LSI-GA, and compares them with the ones generated by the ideal IR process and the ones used in previous work [195]. As we can see for both the dataset the pre-processing steps chosen by LSI-GA are the identical to those of the ideal/optimal IR: both used the same term extraction process, the same term splitting, the same stop-word list and stop-word functions, the same stemming algorithm and the same term weighting schema. For the choice of the number of concepts/topics to be used when applying LSI we observe that the k values provided by LSI-GA are not identical to those utilized by the ideal IR process. Hence

we can conclude that LSI-GA allows us to instantiate an IR process that is close to the ideal one.

3.6.4 Scenario III: Duplicated Bug Report

For this scenario, we use LSI to compute the textual similarity between new bug reports and existing bug reports using their description. We used two different corpora, referred as *Short* and *2ShortLong* (as suggested by [161]). In the former case, each bug is characterized by only the bug title, while in the latter we used both the title and the bug description (the title is weighted twice as much as the bug description). Note that in both cases we combined textual information with information belonging to execution traces.

Besides the textual similarity, as suggested by Wang *et al.* [161], for each bug report in the analyzed corpus, we also generated an execution trace by following the steps to reproduce the bug (such steps are available in the bug description). Using such information we build a bug-to-method matrix, where each bug represents a column, and each method represents a row. The matrix has binary values, i.e., a generic entry (i,j) is one, if the ith method in the corresponding row appears in the execution trace of the jth bug; zero otherwise. Such information can be used to identify bugs having similar execution traces. Such information can complement textual similarity in order to identify duplicate bug reports, i.e., bugs having similar execution traces are candidate to be the same bug. We then apply LSI also on the bug-to-method matrix and we compute the similarity between each pair of bugs (in terms of execution trace) using the *Execution-information-based Similarity.* The final similarity for each pair of bugs report is given by averaging the textual similarity and the similarity of the execution traces of the two bugs.

The design of our study is based on the study introduced by Wang *et al.* [161], but is different in several important aspects. For example, Wang *et al.* used 220 bug reports of Eclipse 3.0 posted on June 2004 and 44 duplicate pairs as well full execution traces with method signatures. For our evaluation, we used 225 bugs (of the same system and posted in the same period) with 44 duplicate pairs, and marked execution traces without method signatures. For collecting the data, even though we followed the methodology described in their approach, we do not have information about the exact set of bugs used by them. Moreover, the execution traces we collected are most likely different since this was a manually collecting process. In summary, we created a realistic reference approach for Task 3, however this does not fully correspond to the one of Wang *et al.*.

For each duplicate pair, we compute the similarity between the oldest submitted bug (among those two) and the remaining 224 bug report in the corpus. Then, to address $\mathbf{RQ}_1$, we compare the accuracy of identifying duplicated bug report achieved by the IR process assembled by LSI-GA with the accuracy yielded by a "baseline" configuration produced by using the preprocessing of Wang *et al.*, and by applying LSI with an "ad-hoc" number of concepts used in traceability link recovery, i.e., $k = 50\%$ of total number of documents [152]. To address $\mathbf{RQ}_2$ we compared the performances of IR process using different number of topics (for LSI) and different pre-processing steps. In particular, we varied the number of

topic from 10 to 200 with steps 1 for both the eclipse textual corpora. We also applied all the possible combinations of the pre-processing steps experimented in this Chapter. Hence, the total number of trials performed on both the corpora is 27,000. With such an analysis we are able to identify the configuration which provides the best recovery accuracy between all the possible configurations aiming at estimating the ideal configuration. We then compared the performances achieved with such an ideal configuration with the performances achieved with the configuration identified by LSI-GA in order to answer RQ2.

Metrics

In this scenario the performance of an IR process is measures using the *recall rate* (RR) [161, 76]. For each bug report of interest, the list of the other bug reports is ranked according to their similarity to that bug report, RR measures if a duplicated bug report is found in the first t top most positions of the ranked list or not. Formally, this metric can be defined as:

$$RR_t = \frac{N_{\text{recalled}}}{N_{\text{total}}}$$

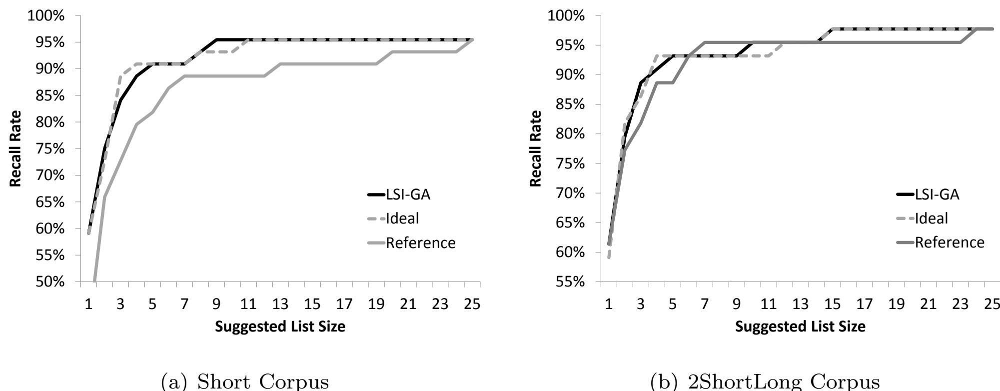
where N_{recalled} is the number of duplicate bug repots were found in first t position of the suggested bug reports list, and N_{total} is the total number of duplicated bug report to be found. We varied the suggested list size t from 1 to 25 and we stored the corresponding RR_t values. Higher RR_t reveals a better ability of an IR process to find duplicated bug report for the suggested list size t. We also used a statistical test to verify whether the RR_t values obtained by one method m_i (e.g., LSI with a specific configuration) is statistically significantly lower than RR_t values yielded by another method m_j (e.g., LSI with another specific configuration). In other words, we want to test the following null hypothesis:

> *There is no statistically significant difference between the RR_t values obtained by m_i and m_j*

The dependent variable of our study is represented by the RR_t values reached by a method for each duplicated bug to be identified. Since the number of duplicated bug is the same for each instantiation of an IR process (i.e., the data were paired), we used the Wilcoxon Rank Sum test [226]. In all our statistical tests we consider p-values < 0.05 as statistically significant. Finally, the p-values are adjusted using Holm's procedure.

Empirical results

Figure 3.8 reports the recall rate for the results produced by using different configurations, i.e., LSI-GA, ideal and a reference configuration. The reference configuration is obtained considering: (i) k equals to half the number of documents [152]; and (ii) by applying a standard corpus preprocessing that is typical to bug duplicates [161] and other SE tasks [195]. For both corpora (*Short* and *2ShortLong*), LSI-GA achieved approximately the same RR values as the ideal. Moreover, LSI-GA produces better results in terms of RR as compared



(a) Short Corpus

(b) 2ShortLong Corpus

Figure 3.8: Recall Rate graphs for Eclipse, with suggested list size raging between 1 and 25.

Table 3.9: Comparison of bug report duplication: the results of the Wilcoxon test.

	Short Corpus	**2ShortLong Corpus**
LSI-GA < Ideal	1	0.66
LSI-GA < Reference	< **0.01**	0.22
Ideal < Reference	< **0.01**	0.64

to the reference. In particular, when the corpus consists of bug report titles only (i.e., *Short*), the RR for LSI-GA is higher than the RR of the reference (i.e., a difference of over 10% for a suggested list size varying from 1 to 19). When the textual corpus is represented by the bug reports titles and their descriptions (i.e., *2ShortLong*), the LSI-GA is generally better (i.e., about 5% improvement) than the reference configuration, except when the suggested list size ranges between 5 and 10. It is interesting to note that, when applying a properly calibrated IR process (such as the one assembled by LSI-GA) on the *Short* corpus, produces approximately the same results as considering the *2ShortLong* corpus without calibration.

Table 3.9 reports the adjusted p-values of the Wilcoxon test for all combinations of the techniques. Results indicate that on the *Short* corpus LSI-GA statistically outperforms the reference. Moreover, there is no significant difference between LSI-GA and the ideal process for the *2ShortLong* corpus. However, the small improvement introduced by LSI-GA over the reference configuration (see Figure 3.8) is not statistically significant. Also, there is no statistical difference when comparing LSI-GA or the reference with the ideal configuration.

Table 3.10 reports the different IR pre-processing steps and the k values for LSI found by the LSI-GA, and compares them with the ones generated by the ideal IR process and the ones used by the baseline. We can observe a scenario that is quite similar to the ones obtained for traceability recovery and feature location: in all the cases the pre-processing steps chosen by LSI-GA and the ideal/optimal IR are identical. The only difference between them is represented by k, i.e., the number of concepts/topics used for LSI. For the *Short*

Table 3.10: Comparison of different IR processes provided by LSI-GA, ideal and reference on Eclipse.

Corpora	Method	k	Special Chars	Digit Chars	Term Splitting	Stopword List	Stopword Functions	Stemming	Weight Schema
Short	Ideal	169	Remove	Include	Camel Case	Yes	No	Porter	tf-idf
	LSI-GA	172	Remove	Include	Camel Case	Yes	No	Porter	tf-idf
	Reference	112	Remove	Remove	Camel Case	Yes	Yes	Porter	tf-idf
2ShortLong	Ideal	180	Remove	Include	Camel Case	Yes	No	Porter	tf-idf
	LSI-GA	179	Remove	Include	Camel Case	Yes	No	Porter	tf-idf
	Reference	112	Remove	Remove	Camel Case	Yes	Yes	Porter	tf-idf

corpus LSI-GA selected $k = 172$ latent concepts/topics while the ideal configuration has $k = 169$ latent concepts/topics. For the *2ShortLong* corpus LSI-GA selected $k = 179$ latent concepts/topics against 169 latent topics of the ideal configuration. Hence, we can conclude that LSI-GA allows us to instantiate an IR process that is close to the ideal one.

3.7 Empirical Evaluation of LDA-GA

This section describes in details the design and the results of the empirical study we conducted to evaluate the proposed LDA-GA approach in the context of software engineering tasks. The study was conducted following the Goal-Question-Metric paradigm by Basili *et al.*[225]. Raw data and working data sets are available for replication purposes[4].

3.7.1 Research Questions

The *goal* of the study is to evaluate LDA-GA in the context of three software engineering tasks. The *quality focus* was ensuring better recovery performances, while the *perspective* was both (i) of a researcher, that wants to find an optimal calibration of the LDA parameter; and (ii) of a project manager, that might be interested to adopt the proposed approach for solving specific SE tasks. The *context* of the study consists of three SE tasks, namely traceability links recovery, feature location, and source code labeling. Specifically, the study aims at addressing the following research questions (RQs) that have been addressed in the context of the three different SE tasks considered in our study:

- **RQ$_1$**: *What is the impact of the configuration parameters on LDA's performance in the context of software engineering tasks?* This research question aims at justifying the need for an automatic approach that calibrates LDA's settings when LDA is applied to support SE tasks. For this purpose, we analyzed a large number of LDA configurations for three software engineering tasks. The presence of a high variability in LDA's performances indicates that, without a proper calibration, such a technique risks being severely under-utilized.

[4]http://www.distat.unimol.it/reports/LDA-GA

Table 3.11: Characteristics of the systems used for traceability recovery.

System	Description	Artifacts		
		Type	Number	Total
EasyClinic	A software system used to manage a doctor's office developed by students	Use Cases Code Classes	30 47	77
eTour	An electronic tourist guide developed by students	Use Cases Code Classes	58 116	174

- **RQ$_2$**: *Does LDA-GA, our proposed GA-based approach, enable effective use of LDA in software engineering tasks?* This research question is the main focus of our study, and it is aimed at analyzing the ability of LDA-GA to find an appropriate configuration for LDA, which is able to produce good results for specific software engineering tasks.

We address both research questions in three different scenarios, representative of SE tasks that can be supported by LDA: traceability link recovery, feature location, and software artifact labeling. LDA was previously used in some of these tasks [10, 173, 220].

3.7.2 Scenario I: Traceability Recovery

The context of this scenario consists of different software artifacts from two projects, namely: EasyClinic and eTour. Both systems were developed by the final year Master's students at the University of Salerno (Italy). The documentation, source code identifiers, and comments for both systems are written in Italian. Table 3.11 reports the characteristics of the considered software systems in terms of type, number of source and target artifacts. Other than the listed artifacts, each repository also contains the traceability matrix built and validated by the application developers. For different kinds of artifacts, the traceability matrices were developed at different stages of the development (e.g., requirement-to-code matrices were produced during the coding phase). We consider such a matrix as the *oracle* to evaluate the accuracy of the different LDA configurations.

To answer to our research questions, we used LDA with a specific input parameters configuration to recover traceability links between artifact pairs on the term-by-document matrices. In all cases the term-by-document matrix is extracted following all the steps of the process described in Section 3.2. Specifically, we used the following pre-process:

- *Term extraction*: each software artifact is pre-processed in order to filter non-textual tokens (i.e., operators, special symbols, some numerals, etc.). The identifiers composed of two or more words are separated into their constituent words using a tool that relies on coding conventions.

- *Stop-word removal*: we used a stop word lists for Italian language. Such a list included, other than Italian standard stop word lists, (i) programming language (C/Java) keywords, (ii) recurring words in document templates (e.g., use case, requirement, or test case template) and (iii) author names.

- *Stemming*: we used Italian Snowball stemmer to reduce in inflected words to their stem, base or root form.
- *Term weighting*: we used the $tf - idf$ which is one of the most widely used weighting scheme [167].
- *Distance measure*: we used the Hellinger distance function which is designed for probabilistic models, like LDA [167].

To answer both the research questions we performed two different traceability recovery activities:

- A_1: recovering traceability links between use cases and source code classes for EasyClinic. The total number of correct links is 83 while the number of all possible links is 1,410.
- A_2: recovering traceability links between use cases and source code for eTour. The total number of correct links is 246 while the number of all possible links is 6,728.

To address $\mathbf{RQ}_1$, we compared the accuracy of recovering traceability links using different configurations for LDA. Specifically, we varied the number of topics from 10 to 100 with step 10 on EasyClinic, and from 10 to 200 with step 10 on eTour. We varied α and β from 0 to 1 with 0.1 increments, and we exercised all possible combinations of such values. We fixed the number of iterations to 500, which resulted to be a sufficient number of iterations for the model to converge. Thus, the total number of trials performed on EasyClinic and eTour were 1,000 and 2,000, respectively. Clearly, although combinatorial, such an analysis is not exhaustive, as it considers a discrete set of parameter values and combinations.

For $\mathbf{RQ_2}$, we compared the accuracy achieved by LDA when the configuration is determined using LDA-GA with (i) the best accuracy achieved by LDA (determined when answering RQ_1) and (ii) the accuracy achieved by LDA on the same system in the previously published studies where an "ad-hoc" configuration was used [219]. While the former comparison is more of a sanity check aimed at analyzing the effectiveness of the GA in finding a near-optimal solution, the latter comparison was aimed at analyzing to what extent LDA-GA is able to enrich the effectiveness and usefulness of LDA in the context of traceability link recovery when properly calibrated.

Metrics

To answer to our research questions, we evaluated LDA's recovery accuracy using we used two well-known IR metrics: precision and recall [167]. Recall measures the percentage of links correctly retrieved, while precision measures the percentage of links retrieved that are correctly identified. A common way to evaluate the performance of retrieval methods consists of comparing the precision values obtained at different recall levels. This result is a set of recall/precision points which are displayed in precision/recall graphs. In order to provide

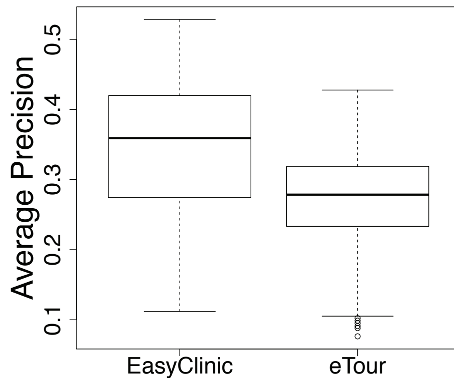


Figure 3.9: Variability of performance achieved by LDA configurations for traceability link recovery

a single value that summarizes the performance, we use the *average precision* which is the mean of the precision scores obtained for each correct link [167].

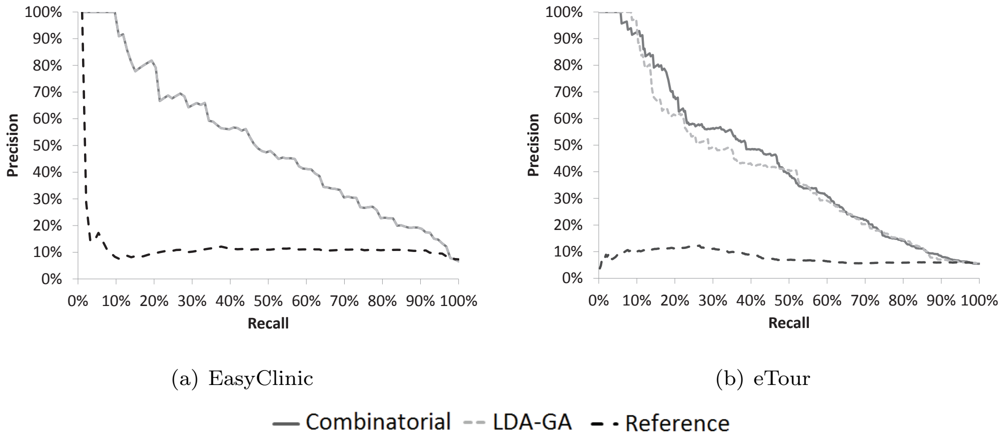
To provide statistical support to $\mathbf{RQ}_2$ we used a statistical test to verify whether the number of false positives retrieved by one method is statistically significantly lower than the number of false positives retrieved by another method similarly as done in previous work [9, 16, 17]. In other words, we compared the false positives retrieved by method m_i (e.g., LDA with a specific configuration) with the false positives retrieved by method m_j (e.g., LDA with another specific configuration) to test the following null hypothesis:

> *There is no statistically significant difference between the number of false positives retrieved by m_i and m_j*

The dependent variable of our study is represented by the number of false positives retrieved by the traceability recovery method for each correct link identified. Since the number of correct links is the same for each traceability recovery activity (i.e., the data were paired), we used the Wilcoxon Rank Sum test [226]. In all our statistical tests we consider p-values $<$ 0.05 as statistically significant.

Empirical results

To answer to $\mathbf{RQ}_1$, Figure 3.9 shows boxplots summarizing the average precision values obtained using the 1,000 and 2,000 different LDA configurations on EasyClinic and eTour, respectively. We used these boxplots to highlight the variability of the average precision values across different configurations. As shown, the variability of LDA's performance is high: the average precision ranges between 11% and 55% on EasyClinic and between 7% and



(a) EasyClinic

(b) eTour

Figure 3.10: Traceability recovery: precision/recall graphs.

43% on eTour. For EasyClinic, more than 75% of the different LDA configurations obtained an average precision lower than 45% (see first three quartiles in Figure 3.9). Moreover, only a small percentage of the configurations executed in the combinatorial search (about 3.6%) obtained an average precision greater than 50%. In the end, only one of them achieved the highest value, 52%. Similarly for eTour, the configurations placed in the first three quartiles (about 75% of the set) obtained an average precision lower than 40%, while less than 1% of the total amount of executed configurations in the combinatorial search (2,000 configurations) achieved an average precision greater than the 40%. Only one configuration achieved the highest average precision (47%).

In summary, for $\mathbf{RQ}_1$ we can assert that for traceability recovery, LDA shows high variability. Thus, LDA's efficiency for establishing links between software artifacts depends on the particular configuration $P = [n, k, \alpha, \beta]$ used to derive latent topics. Indeed, "bad" configurations can produce poor results while "optimal" configurations (which represent a small portion of all possible LDA configurations) can lead to very good results.

For what concerns $\mathbf{RQ}_2$, Figure 3.10 reports the precision/recall graphs obtained by LDA using (i) the best configuration across 1,000 and 2,000 different configurations executed in the combinatorial search; (ii) the configuration identified by LDA-GA; and (iii) an "ad-hoc" configuration used in a previous study where LDA was used on the same repositories [80]. For both EasyClinic and eTour, LDA-GA was able to obtain a recovery accuracy close to the accuracy achieved by the optimal configuration across 1,000 and 2,000 different configurations executed in the combinatorial search. In particular, for EasyClinic LDA-GA returned exactly the configuration identified by the combinatorial search (i.e., the two curves are perfectly overlapped) while on eTour the two curves are comparable. Moreover, the average precision achieved by the configuration provided by LDA-GA is about 41%, which is comparable with the average precision achieved with the optimal configuration, which is about 43% (only a

Table 3.12: Results of the Wilcoxon test for traceability recovery.

	EasyClinic	**eTour**
LDA-GA < Combinatorial	1	1
LDA-GA < Oliveto *et al.* [80]	< **0.01**	< **0.01**
Combinatorial < Oliveto *et al.* [80]	< **0.01**	< **0.01**
Combinatorial < LDA-GA	1	< **0.01**

Table 3.13: Characteristics of the systems used in feature location.

System	**KLOC**	**Classes**	**Methods**	**Features**
jEdit	104	503	6,413	150
ArgoUML	149	1,439	11,000	91

small difference of 2%). Among 2,000 different configurations tried for the combinatorial search, only five configurations obtained an average precision comparable or greater than the one achieved by LDA-GA, i.e., the configurations obtained by LDA-GA belong to the 99% percentile for the distribution reported in Figure 3.9. Finally, comparing the performance achieved by LDA-GA with the performance reached by other LDA configurations used in previous work [80], we can observe that the improvement is very substantial for both software systems.

Table 3.12 reports the results of the Wilcoxon test (i.e., the adjusted p-values) for all combinations of the techniques (statistically significant results are highlighted in bold face). As we can see, there is no statistically significant difference between the performance obtained by LDA-GA and the combinatorial search for EasyClinic. However, for eTour the combinatorial search performs significantly better than LDA-GA. However, considering the precision/recall graph reported in Fig. 3.10, we can observe that the difference is relatively small.

3.7.3 Scenario II: Feature Location

The context of this scenario is represented by two software systems, *jEdit v4.3* and *ArgoUML v0.22*. jEdit is an open-source text editor for programmers, while ArgoUML is a well known UML editor. Table 3.13 reports the characteristics of the considered software systems in terms of number of classes, number of methods, as well as KLOC and the number of features to be located. These software systems have been used in previous studies on feature location [215, 228]. For these systems are available (i) the gold sets (i.e., mappings between source code and features) at method level granularity, (ii) the description of the features from bug reports and (iii) execution traces. For jEdit the gold sets were generated via analysis of the SVN commits submitted between releases 4.2 and 4.3. The issue identifiers from the SVN logs were extracted, and were mapped to issues from the issue tracking system. On the other

hand, the changes from the SVN commits were mapped to the methods from the source code that were modified by that commit [193]. Similar analysis was performed to generate the gold sets for ArgoUML version 0.22.

To answer $\mathbf{RQ}_1$, we compared the effectiveness measure of LDA using different configurations. Specifically, we varied the number of topics from 50 to 500 with step 50 for both ArgoUML and jEdit. We varied α and β from 0 to 1 with 0.1 increments. Similarly to the traceability task, we fixed the number of iterations to 500. We exercised all possible combinations of such values. Thus, the total number of trials performed on both software systems consisted of 1,000 different LDA combinations. For $\mathbf{RQ}_2$, similarly to the previous scenario, we compared the performance achieved by LDA-GA with (i) the best performance achieved by LDA when answering $\mathbf{RQ}_1$ and (ii) the performance obtained by LDA using the *source locality* heuristic proposed by Grant and Cordy for the feature location task [211].

Metrics

The performance of LDA in this scenario was analyzed using the *effectiveness measure (EM)* [73]. Given a feature of interest, this measure estimates the number of methods a developer needs to inspect before finding a method relevant to that feature in the ranked list. The goal of each technique is to assist the developer such that she needs to look at fewer methods. The effectiveness can be regarded as an inverse measure: the lower the rank of the first method is, the better the result becomes [73].

A common way to evaluate the performance of feature location methods consists of comparing the distribution of the EM values. This result is a set EM values (each one for each feature) which are displayed through boxplots. In order to provide a single value that summarizes the performance, we use the *average effectiveness measure*, that can be defined as the mean of the EM scores obtained for each feature to locate [73].

To provide statistical support to $\mathbf{RQ}_2$ we used a statistical test to verify whether the EM values obtained by one method m_i (e.g., LDA with a specific configuration) is statistically significantly lower than EM values yielded by another method m_j (e.g., LDA with another specific configuration) similarly as done in previous work [73, 193]. In other words, we want to test the following null hypothesis:

> *There is no statistically significant difference between the EM values obtained by m_i and m_j*

The dependent variable of our study is represented by the EM values reached by the feature location method for each feature to be located. Since the number of feature location is the same for each feature location (i.e., the data were paired), we used the Wilcoxon Rank Sum test [226]. In all our statistical tests we consider p-values < 0.05 as statistically significant.

Empirical results

Figure 3.11 shows the boxplots summarizing the variability of the average effectiveness measure (EM) values obtained using 1,000 different LDA configurations. As in the previous task,

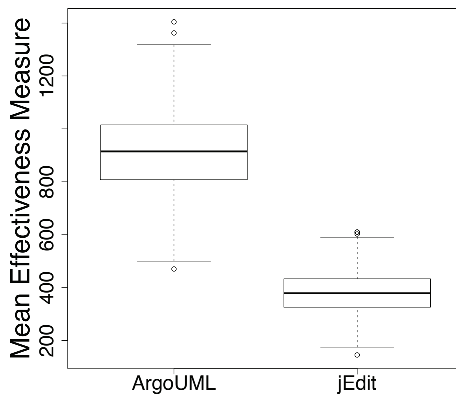
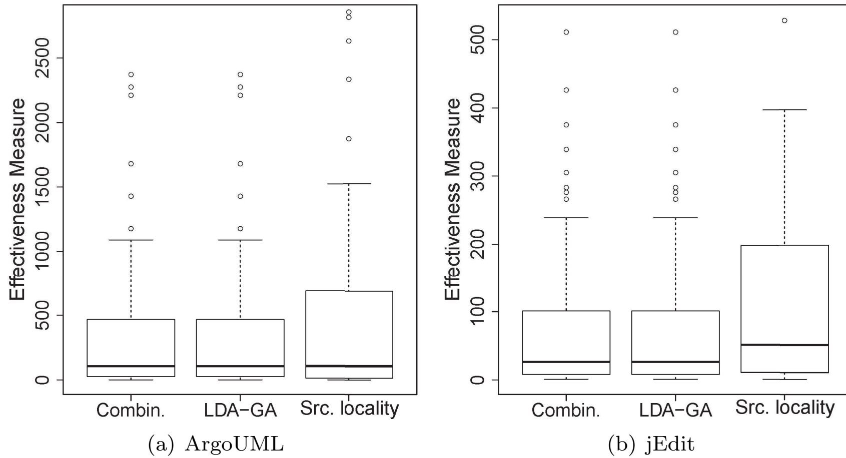


Figure 3.11: Variability of performance achieved by LDA configurations for feature location.

the feature location results show high variability in their EM, which ranges between 472 and 1,416 for ArgoUML and between 145 and 600 for jEdit. For ArgoUML, we observed that more than 90% of different configurations produced an average EM ranging between 600 and 1,200, while only a small percentage (about 3%) produced an optimal average EM lower than 600. Within this small number of optimal configurations only one configuration obtains the lowest (i.e., the best) EM of 472. Similarly, for jEdit, 95% of different configurations produced an average EM that ranges between 200 and 1,600, while only one achieved the smallest average EM of 148. These results for $\mathbf{RQ}_1$ suggest that without a proper calibration, the performance of LDA risks of being unsatisfactory.

For $\mathbf{RQ}_2$, Figure 3.12-a shows boxplots for ArgoUML of the EM values achieved by three different configurations: (i) the best configuration obtained by a combinatorial search across 1,000 different LDA configurations (combinatorial search); (ii) the configuration obtained using LDA-GA; and (iii) the best configuration obtained using the source locality heuristic [211]. First, we can note that the configuration obtained via LDA-GA is exactly the same as the one obtained from the combinatorial search, thus LDA-GA was able to find the best configuration (i.e., with the lowest average EM). Comparing the performance of LDA-GA with the source locality heuristic, we can observe that for the first two quartiles, there is no clear difference (the median values are 107 and 108 for LDA-GA and source locality heuristic respectively). Considering the third and fourth quartiles, the difference becomes substantial: the third quartile is 467 for LDA-GA and 689 for the previous heuristic, while for the fourth quartiles we obtained 4,603 for LDA-GA and 7,697 for source locality heuristic. Overall, LDA-GA reached an average EM equal to 473, as opposed to EM equal to 707 obtained using the source locality heuristic.

Boxplots for jEdit are shown in Figure 3.12-b. It shows the boxplots of the effectiveness measure values achieved by three different configurations, two of them derive from *a priori*



(a) ArgoUML

(b) jEdit

Figure 3.12: Box plots of the effectiveness measure for feature location for ArgoUML and jEdit.

Table 3.14: Results of the Wilcoxon test on feature location performances.

	jEdit	**ArgoUML**
LDA-GA < Combinatorial	0.09	1
LDA-GA < Source Locality Heuristic [211]	**0.02**	**0.02**
Combinatorial < Source Locality Heuristic [211]	**0.02**	**0.02**

approach (LDA-GA and the heuristic by Grant and Cordy) while the remaining one is the configuration obtained by an *a posteriori* search consisting in the best across 1,000 different configurations aimed at find the global optimum. First, we can note that the configuration by LDA-GA close to the best configuration obtained by the combinatorial search, thus the proposed approach turned out to be able to find a near optimal configuration in terms of effectiveness measure. Comparing the performance of the proposed approach with that obtained by a previous heuristic, we can observe that for all the quartiles there is a clear difference: the median ($q_{2/4}$) for LDA-GA is 26.5 while for the heuristic is 51.5. The third quartile $q_{3/4}$ is 98.5 for LDA-GA against $q_{3/4} = 197$ of Grant-Cordy, while for the fourth quartile we obtained $q_{4/4} = 3,297$ for LDA-GA and $q_{4/4} = 3,372$ for Grant-Cordy. Such an analysis is confirmed by the average effectiveness measure: for LDA we reached an average EM equals to 154, against 234 obtained by the Grant and Cordy's heuristic.

Wanting to provide a statistical support to our preliminary results, the Wilcoxon Rank Sum test was used to test if the difference between the effectiveness measures of two feature location techniques is statistically significant or no. We also verify whether there is or no significant difference with the best configuration obtained by the combinatorial search as sanity check. Table 3.14 reports the results of the Wilcoxon test (i.e., the adjusted p-values)

Table 3.15: Characteristics of the systems used in source code labeling.

System	KLOC	Classes	Sampled classes
JHotDraw	29	275	10
eXVantage	28	348	10

for all combinations of the techniques (statistically significant results are shown in bold face). As we can see, there is no statistical difference between the performance obtained by LDA-GA and the combinatorial search. Based on the results of the statistical tests, we can assert that LDA-GA is able to find the optimal or the near-optimal configurations. Moreover, LDA-GA significantly outperforms the previously published source locality heuristic (p-value< 0.02).

3.7.4 Scenario II: Source Code Labeling

In this scenario, we used LDA to automatically "label" source code classes using representative words. Specifically, we extracted topics from a single class (using LDA), and then we ranked all the words characterizing the extracted topics according to their probability in the obtained topic distribution. The top 10 words belonging to the topic with the highest probability in the obtained topic distribution were then used to label the class [10].

The *goal* of the study is to analyze whether the LDA configuration obtained by LDA-GA improves the performances of LDA-based automatic source code labeling. The *quality focus* was ensuring better labeling performances, while the *perspective* was both (i) of a researcher, that wants to find an optimal calibration of the LDA parameter; and (ii) of a project manager, that might be interested to adopt the proposed approach to summarize the source code.

The study was conducted on 10 classes from JHotDraw and 10 classes from eXVantage. The former is an open-source drawing tool, and the latter is a novel testing and generation tool. Their characteristics are summarized in Table 3.15. For the sampled classes, we had user-generated labels from a previously published work [10], and these represented our "ideal" labels. Specifically, the gold sets are obtained through an experiment in which we asked 38 subjects[5] to describe classes taken from the two software systems using at most ten words extracted from the source code and comments. Finally, for each class we consider as gold set the set of words that were used by at least the 50% of the subjects[6]. Then, we analyzed to what extent the highest weighted words indexed using various LDA configurations overlap with those identified by humans.

Also in this scenario, in order to address $\mathbf{RQ}_1$ we compared the recovery accuracy of LDA using different settings. Specifically, we varied the number of topics from 10 to 50 with step 10 for both JHotDraw and eXVantage. As for α and β, we varied them between 0 and 1 by increments of 0.1. We fixed the number of iterations to 500 as in the previous two tasks. We

[5]The subjects were selected among the Bachelor's Students in Computer Science of the University of Molise and the Master's Student in Computer Science of the University of Salerno.

[6]Further details on this controlled experiment can be found in [1].

exercised all possible combinations of such values. Thus, the total number of trials performed on JHotDraw and eXVantage was 500 on both systems.

For **RQ**$_2$, we compared the accuracy achieved by LDA-GA with (i) the best accuracy achieved by LDA while iterating through the parameters and (ii) the accuracy achieved by LDA reported by De Lucia *et al.* [10, 1].

Metrics

As overlap measure we used the asymmetric Jaccard measure [167]. Formally, let $K(C_i) = \{t_1, \ldots, t_m\}$ and $K_{m_i}(C_i) = \{t_1, \ldots, t_h\}$ be the sets of keywords identified by subjects and the technique m_i, respectively, to label the class C_i. The overlap was computed as follows:

$$overlap_{m_i}(C_i) = \frac{|K(C_i) \cap K_{m_i}(C_i)|}{K_{m_i}(C_i)}$$

Note that the size of $K(C_i)$ might be different from the size of $K_{m_i}(C_i)$. In particular, while the number of keywords identified by LDA is always 10 (by construction we set $h = 10$), the number of keywords identified by subjects could be more or less than 10 (generally it is 10, but there are few cases where the number is different). For this reason, we decided to use the asymmetric Jaccard to avoid penalizing too much the automatic method when the size of $K(C_i)$ is less than 10.

Clearly, each selected class has an own overlap value; thus, each automated labeling technique provides different values whose number is equal to the number of classes. In order to provide a single value that summarizes the performance, we use the *average overlap value* (AO), that can be defined as the mean of the *overlap values* obtained for each class to be labeled.

Empirical results

For **RQ**$_1$, Figure 3.13 shows boxplots for the average percentage overlap (AO) values obtained using 500 different LDA configurations. Even if in this case the corpus of documents (the total number of classes and the vocabulary size) is really small, as compared to the size of the repository considered for the other tasks, LDA also shows a high variability of performances, ranging between 18% and 66% on JHotDraw, and between 13% and 77% on ExVantage. For JHotDraw, it can be noted how, more than 72% of the different configurations obtained an AO value ranging between 25% and 55%, while only a small percentage (about 1%) obtains an optimal AO greater than 60%. Within this small number of optimal configurations, only one achieves the highest AO of 64%. Similarly, for ExVantage the majority (about 79%) of the different configurations obtained an AO ranging between 10% and 70%, while only one configuration achieved the highest AO of 77%.

For **RQ**$_2$, Table 3.16 reports the statistics of the overlap between the user-based labeling and the automatic labeling obtained using (i) LDA-GA; (ii) the best configuration achieved using the combinatorial search, i.e., the configuration which has the higher percentage overlap

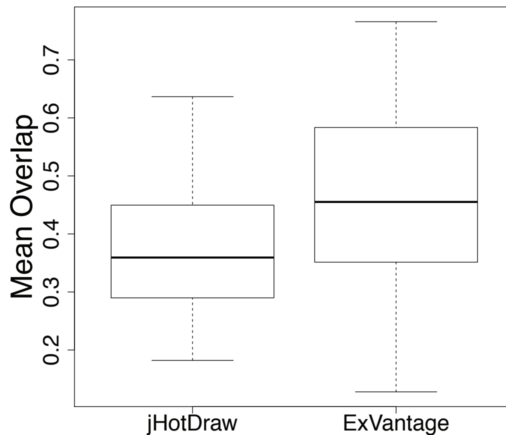


Figure 3.13: Variability of performance achieved by LDA configurations for labeling.

among 500 different configurations; and (iii) the LDA configuration used in the previous work [10] for the same task. For both systems, LDA-GA obtains a percentage overlap with the user labeling that is close to the combinatorial search, with a difference from the best LDA configuration (obtained by the combinatorial search) of about 3% for ExVantage and 1% for JHotDraw. For ExVantage, among the 500 different LDA configurations computed in the combinatorial search, only 12 configurations have an average overlap greater or equal to 74.33%. We can also observe that there are only small differences for the median and second quartile between LDA-GA and the global optimum, while for the other quartiles there is no difference. Similarly, among 500 different configurations evaluated for JHotDraw, only one configuration is comparable with LDA-GA. By comparing the quartile values obtained for JHotDraw, we can note that the difference between LDA-GA and the combinatorial search optimum is about 2%-3% on average. Finally, we can observe how the performances of LDA configured using LDA-GA are significantly better than those reported in the previous work [10] (where α and β were set to default of $50/k$ and 0.1 respectively). For ExVantage we obtain an improvement in terms of mean overlap of about 14-20%, while for JHotDraw we get an improvement of about 5-6%.

3.8 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. For the investigated software engineering tasks, we evaluated the performances of LSI-GA, LDA-GA and alternative approaches using well-established metrics, namely precision and recall, effectiveness measure and Recall Rate, and oracles already used and validated in previous

Table 3.16: Average Overlap between automatic and manual labeling.

exVantage					
	LDA		**De Lucia et al. [10]**		
	LDA-GA	**Combinatorial**	**n = M**	**n = M/2**	**n = 2**
Max	**100%**	**100%**	100%	100%	100%
3rd Quartile	**95%**	**95%**	71%	70%	69%
Median	67%	**70%**	59%	60%	54%
2nd Quartile	60%	**67%**	34%	50%	41%
Min	**50%**	**50%**	0%	0%	40%
Mean	74%	77%	52%	56%	60%
St. Deviation	19%	17%	31%	34%	23%

JHotDraw					
	LDA		**De Lucia et al. [10]**		
	LDA-GA	**Combinatorial**	**n = M**	**n = M/2**	**n = 2**
Max	**100%**	**100%**	100%	100%	100%
3rd Quartile	81%	**82%**	73%	70%	66%
Median	71%	**75%**	65%	61%	56%
2nd Quartile	47%	**50%**	46%	45%	41%
Min	14%	14%	0%	38%	**29%**
Mean	65%	66%	59%	60%	59%
St. Deviation	28%	26%	28%	20%	24%

studies [215, 228, 229, 219, 161]. Finally, we used as baseline of comparison performances achieved using IR processes and calibrations used in previous papers. As for detecting duplicate bug reports, it was not possible to fully replicate the approach of Wang *et al.* [161] due to the unavailability of all the required information. However, subsection 3.6.4 explains the details and the rationale for using a baseline for comparison for such a task. For the labeling task, we compared LDA-based labeling with a user-generated labeling, using, again, the dataset previously verified and published [10].

Threats to *internal validity* are related to co-factors that could have influenced our results. We limited the influence of GA randomness by performing 30 GA runs, and considering the configuration achieving the median performance. Threats to *conclusion validity* concern the relationship between treatment and outcome. To support our claims, we used non-parametric statistical tests (i.e., Wilcoxon rank sum test).

Threats to *external validity* concern the generalization of our results. It is highly desirable to replicate the studies carried out on the experimented scenarios on other datasets. Secondly, we consider only a subset of the possible treatments for the various phases such as term extraction, stop words removal, stemming, and term weighting. Although the chosen treatments are well representative of most of the ones used in literature, it is worthwhile to investigate further possibilities.

3.9 Conclusion and Future Work

The application of IR techniques to software engineering problems requires a careful construction of a process consisting of various phases, i.e., term extractions, stop word removal, stemming, term weighting, and application of an algebraic IR method. Each of these phases can be implemented in various ways, and requires careful choice and settings, because the performances significantly depend on such choices [210].

This chapter investigated a search-based approach to automatically assemble a (near) optimal IR process when solving software engineering problems such as traceability link recovery or feature location. Noticeably, the proposed approach is unsupervised and task independent, as it evaluates the extent to which the artifacts can be clustered after being processed. We applied the proposed approach—named LSI-GA—to three software engineering tasks, namely traceability link recovery, feature location, and detection of duplicate bug reports. Results of our empirical evaluation indicate that: for traceability recovery and feature location, the IR processes assembled by LSI-GA significantly outperform those assembled according to what previously done in literature. While for duplicate bug report detection, the obtained results do not always significantly improve the performances of the baseline approach, as such a baseline is already close to the ideal optimum. Moreover, in most cases, the performances achieved by LSI-GA are not significantly different from the performances of an ideal IR process that can be built by considering all possible combinations of treatments for the various phases of the IR process, and by having a labeled training set available (i.e., by using a supervised approach).

In this chapter we also proposed a similar search-based approach for calibrating LDA, named LDA-GA. We also conducted several experiments to study the performance of LDA configurations based on LDA-GA with those previously reported in the literature (i.e., existing heuristics for calibrating LDA) and a combinatorial search. The results obtained indicate that (i) applying LDA to software engineering tasks requires a careful calibration due to its high sensitivity to different parameter settings, that (ii) LDA-GA is able to identify LDA configurations that lead to higher accuracy as compared to alternative heuristics, and that (iii) its results are comparable to the best results obtained from the combinatorial search.

Work-in-progress aims at extending the proposed approach in various ways. First, we plan to use a more sophisticated evolutionary algorithm, employing genetic programming (GP) to assemble different phases in different ways, including creating ad-hoc weighting schemata [212] or extracting ad-hoc elements from artifacts to be indexed, e.g., only specific source code elements, only certain parts-of-speech. Also, we plan to also optimize the choice of the most appropriate IR algebraic method, also in cases for which such a method does not imply document clustering. Finally, we plan to incorporate into the adopted solution encoding schema some enhancing strategies that have been proposed in order to improve the performances of IR methods when applied to software engineering task, such as smoothing filters [9, 2], part of speech-tagger [17] and relevance feedback [151, 176].

Chapter 4

Multi-Objective Cross-project Defect Prediction

Contents

4.1 Introduction and Motivation 79

4.2 Background and Related work 80

4.2.1 Metrics 81

4.2.2 Machine Learning Techniques 83

4.2.3 Cross-project defect prediction 84

4.3 MODEP: Multi-Objective Defect Predictor 86

4.3.1 Data Preprocessing 86

4.3.2 Multi-Objective Defect Prediction Models 87

4.3.3 Training the Multi-Objective Predictors using Genetic Algorithms 92

4.4 Design of the Empirical Study 93

4.4.1 Definition and Context 93

4.4.2 Research Questions 94

4.4.3 Variable Selection 95

4.4.4 Analysis Method 96

4.4.5 Implementation and Settings of the Genetic Algorithm 98

4.5 Study Results 99

4.5.1 RQ_1: *How does MODEP perform compared to single-objective prediction?* 99

4.5.2 RQ_2: *How does MODEP perform compared to the local prediction approach?* 107

4.5.3 Benefits of MODEP as Compared to Single-Objective Defect Predictors 109

4.6 Threats to Validity 111

4.7 Conclusion and Future Work 112

4.1 Introduction and Motivation

Finding and fixing defective software components is a very important activity for development organizations in order to increase the reliability of the software itself. However, because of the limited amount of resources (people and time) only some components can be adequately tested and eventually fixed before spreading the new release. Therefore, approaches for defect prediction have been used to identify software modules having a high likelihood to exhibit fault, and that should therefore be better analyzed and tested.

Turban *et al.* [84] and Zimmermann *et al.* [85] found that cross-project defect prediction does not always work well. The reasons are mainly due to the projects' heterogeneity, in terms of domain, source code characteristics (e.g., classes from a project can be intrinsically larger or more complex than those of another project), and process organization (e.g., one project exhibits more frequent changes than another) [86, 87]. Hence, traditional prediction models cannot be applied out-of-the-box for cross-project prediction. Recently, a study by Brahman *et al.* [88] pointed out that cross-project defect predictors do not necessarily work worse than within-project predictors. It depends on the evaluation metrics that are used to analyse the performances of the models. Indeed, even if the cross-project strategy is worst than within-strategy when considering the prediction accuracy, from the cost effectiveness point of view the scenario is quite different: cross-project defect prediction models become quite competitive when considering the inspection cost. Specifically, they allow to pursue a good compromise between the number of defect-prone artifacts that the model predict, and the amount of code—i.e., LOC of the artifact predicted as defect-prone—that a developer needs to analyze/test to discover such defects. However, in [88] the prediction models are built using the same methodology used for traditional classification models, while the cost-effectiveness is considered only when evaluating the performances of the built models.

Stemming from the considerations by Brahman *et al.* [88] this chapter proposes to shift from the single-objective defect prediction model—which recommends a set of likely defect-prone artifacts and tries to minimize the prediction error—towards multi-objective defect prediction models. We use the NSGA-II algorithm [107] to train machine learning predictors. Such an algorithm evolves the coefficients of machine learning models (in our work a logistic regression or a decision tree, but the approach can be applied to other machine learning techniques) to build Pareto fronts of predictors that (near) optimize the two conflicting objectives of cost and effectiveness. In such a context, the cost is represented by the cumulative LOC of the entities (classes in our study) that the approach predict as likely defect-prone. However, without loss of generality, one can also model the testing cost, considering other aspects (such as cyclomatic complexity, number of method parameters, etc) instead of LOC. As effectiveness measure, we use either (a) the proportion of actual defect-prone classes among the predicted ones, or (b) the proportion of defects contained in the classes predicted as defect-prone out of the total number of defects. In essence, instead of training a single model minimizing the prediction error, we obtain a set of predictors, whose performance are Pareto-efficient [230] in terms of cost-effectiveness. Therefore, for a given budget (i.e., LOC that can be reviewed or tested with the available time/resources)

the software engineer can choose a predictor that (a) maximizes the number of defect-prone classes tested (which might be useful if one wants to ensure that an adequate proportion of defect-prone classes has been tested), or (b) maximizes the number of defects that can be discovered by the analysis/testing.

The proposed approach, called MODEP (**M**ulti-**O**bjective **DE**fect **P**redictor), has been applied on 10 datasets from the PROMISE[1]. The results achieved show that (i) the MODEP is more effective than single-objective predictors in the context of a cross-project defect prediction, i.e., it identifies a higher number of defects at the same level of inspection cost; (ii) MODEP provides software engineers the ability to balance between different objectives; and (iii) finally, MODEP outperforms a local prediction approach based on clustering proposed by Menzies *et al.* [86].

The chapter is organized as follows. Section 4.2 summarizes background notions and related work on defect prediction giving particular attention to cross-project strategy. Section 4.3 formulates the problem of finding cost-effective defect prediction models as multi-objective problem and describe the MODEP approach. Section 4.4 describes the empirical study aimed at evaluating MODEP, while Section 4.5 reports the obtained results. Finally, Section 4.6 discusses the threats to validity that could have affected our study. Section 4.7 outlines directions for future work.

4.2 Background and Related work

Even if it is not possible a priori to know where exactly the defects are, there are several factors that can correlate with the likelihood of a software component to be defect-prone. For example, the complexity is one of the most investigated factor, since the defect proneness increases with the number of entities and the number of interaction and dependencies between them. Other factor can be (i) the likelihood of components to violate requirements, (ii) high change rate of requirements and source code, (iii) the quality of the development process [111]. The idea of defect prediction approaches is that by having bug and change databases available it is possible to train mathematical model that automatically relate defects to the possible factors.

A generic defect prediction approach follows the steps shown in Figure 4.1:

- *Data extraction*: the first step consists of extracting predictors (e.g. software metrics) from the software to be predicted (test set). Different predictors can be extracted, such as object-oriented metrics, e.g. the Chidamber and Kemerer (CK) metrics [231], process metrics [232], history based metrics [233, 234], structural metrics [89].

- *Data preprocessing*: once the predictors are extracted, they are preprocessed in order to address the problem of data heterogeneity either to normalize the data.

- *Choosing the training set*: once a set of software metrics is extracted, it is necessary to choose the past data to be used for training machine learning techniques. There

[1] https://code.google.com/p/promisedata/

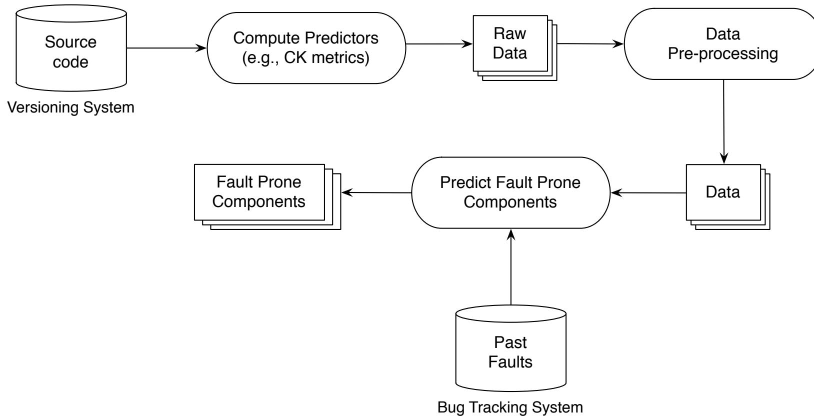


Figure 4.1: Building a defect prediction model.

are two main strategies: the first strategy, named *within-project strategy*, consists of training a classification/regression model on past data of the same software project; with a second strategy, named *cross-project strategy*, the prediction model is trained on past data belonging to different software projects with different domains and it is used to predict the defect proneness of entities belonging to a new project [84, 85, 86, 87].

- *Mathematical model*: in this step a machine learning technique is used to build a prediction model which relates software metrics and defect-proneness of components from the training set.
- *Prediction*: the mathematical model built on training set is then used for predicting the defect proneness of components belonging to the test set.

All previous studies focused on one or more of the steps in figure 4.1. The majority of them focused on (i) finding the software metrics or predictors which correlate with the defect proneness of software components; (ii) proposing/comparing machine learning techniques; (iii) proposing/comparing within-project and cross-project strategies. The most relevant works are described in the next sub-sections.

4.2.1 Metrics

From a design metrics perspective, there have been studies involving the Chidamber/Kemerer (CK) metric suite [231] which can be considered as indicator of product quality. The CK metric suite consists of seven metrics, that are designed for object oriented programs:

- *Lines of code* (LOC): this simple metric measures the number of non-commented lines of code for each software component (e.g., in a class).

- *Weighted Methods per Class* (WMC): it measures the complexity of a class by the number of methods contained in that class including public, private and protected methods.
- *Coupling Between Objects* (CBO): it is equal to the number of classes coupled to a given class. This coupling can occur through class member variables, function parameters, classes defined locally in class member function bodies, return types, and exceptions.
- *Depth of Inheritance* (DIT): it measures the maximum class inheritance depth for a given class.
- *Number Of Children* (NOC): it measures the number of classes inheriting from a given parent class.
- *Response For a Class* (RFC): it measures the number of methods that can be invoked for an object of given class.
- *Lack of Cohesion Among Methods* (LCOM): it counts the sets of methods in a class that are not related through the sharing of some of the class's fields.

The CK metric suite was used by Basili *et al.* [235] to predict the fault-proneness of eight student projects, by Briand *et al.* [236] for an industrial case study. However, further predictors have been proposed and investigated in literature. Moser *et al.* [232] considered process metrics for defect prediction including code churn, past bugs and refactoring, number of authors, file size and age, etc. Other works on process metrics were conducted by Ostrand *et al.* [233] and Kim *et al.* [234]. Arisholm *et al.* [89] proposed twelve structural metrics. Shin *et al.* [237] investigated the usefulness of the call structure of the program and found that it provided noticeable improvement in prediction accuracy, but only marginally improved the best model based on history and non-calling structure code attributes. Marcus *et al.* [157] proposed new measure for the cohesion of classes in object-oriented software systems based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. Specifically, they used the textual similarity between classes computed using LSI for defect prediction on several C++ systems, including Mozilla. Arnaoudova *et al.* [238] argued that the use of the same identifiers in different contexts may increase the risk of faults. Thus, they investigated this conjecture using a measure combining term entropy and term context coverage to study whether certain terms increase the odds ratios of methods to be fault-prone. Pinzger *et al.* [239] empirically investigated the relationship between the fragmentation of developer contributions and the number of post-release failures. They represented developer contributions with a developer-module network called contribution network. Then, they used network centrality measures to measure the degree of fragmentation of developer contributions. Wolf *et al.* [240] analyzed the network of communications between developers to understand how they are related to defect proneness of software components. Their results indicate that developer communication plays an important role in the quality of software integrations. Bacchelli *et al.* [241] integrated different

repositories, such as configuration management systems, bug tracking systems, and mailing lists to trace the evolution of software. They introduce metrics that measure the popularity of source code artifacts, i.e. the amount of discussion they generate in e-mail archives, and showed a significant correlation to the defects found in the system. An extensive analysis of the different software metrics and approaches used for defect prediction can be found in the survey by D'Ambros *et al.* [242].

4.2.2 Machine Learning Techniques

Several machine learning techniques have been used in literature for defect prediction, such as logistic regression [235, 243, 83], Support Vector Regression (SVR) [244], Radial Basis Function Network (RBF Network) [245], multi-layer perceptron (MLP) [246], Bayesian network [247, 248], decision trees [249] and decision tables [250]. For example, Basili *et al.* [235], Gyimothy *et al.* [243] and by Nagappan *e*t al. [83], used logistic regression to relate product metrics to class defect-proneness for within-project strategy. Recently, it has been also used in several works on cross-project defect prediction, as for example by Zimmermann et al. [85] which is one of the earliest work on cross-project strategy and by Brahman *et al.* [88]. Zimmermann and Nagappan [244] used Support Vector Machines (SVM) using linear, polynomial, and radial basis function (RBF) as kernels. Ceylan *et al.* [246] considered another neural network model, called Multilayer Perceptron (MLP), with the principal component analysis for dimensionality reduction. Felton *at al.* [251] suggested to use Bayesian networks for defect prediction showing how Bayesian networks provide accurate results for software quality for real world projects. In another study by Felton *at al.* [247] Bayesian Networks were used for incorporating empirical data and expert judgement in a combined approach. Okutan and Yildiz [248] used Bayesian networks to train models on the set of software metrics used in Promise data repository [248].

A general prediction model can be viewed as a function $F : \mathbb{R}^n \rightarrow \mathbb{R}$, which takes as input a set of predictors and returns a scalar value—ranging within the interval $[0; 1]$—that measures the likelihood that a specific software entity is defect-prone. Thus, a defect prediction model uses a training set (that come from the same project either other projects) with various predictors of the available software entities (files, modules, binaries etc.) as independent variables ($\mathbb{R}^n$) and return the corresponding defect proneness. The prediction model which best fits this training set (or equivalently the model which minimize the prediction error) is then used to predict the future defect proneness of new the entities via the set of the corresponding predictors. The estimated defect proneness of new entities is then used to rank such entities in order to identify the most defect-prone entities to be investigated in quality assurance activities. In some cases a fixed cut point μ is used to cut the ranked list and provide the software engineer with the top-μ ranked entities (these entities are the estimated defect prone entities). The difference between the various machine learning techniques depends on the specific function F that is used to map the predictors to the estimated defect proneness.

For example, with multivariate logistic regression model the function F is the *logit* func-

tion which takes as input a set of predictors and returns a scalar value which measures the estimated defect proneness of software modules taken in consideration. With the Radial Basis Function (RBF) Network [252], the function F is Gaussian function used to activate the hidden layer of the corresponding neural network[2].

4.2.3 Cross-project defect prediction

As any supervised prediction approach, defect prediction models require the availability of enough data about defects occurred in the history of a project, as also pointed out by Nagappan *et al.* [83]. For this reason, such models are difficult to be applied on new projects for which limited or no historical defect data is available. To overcome this limitation, several authors [84], [85], [86], [87] have suggested the application of *cross-project* strategy for defect prediction, i.e., using data from other projects to train machine learning models, and then perform a prediction on a new project. The earliest works on such a topic provided empirical evidence that simply using projects in the same domain does not help build accurate prediction models [85, 253]. For this reason, Zimmermann *et al.* [85] identified a series of factors that should be evaluated before selecting the projects to be used for building cross-project predictors. However, also using such guidelines the choice of the training set is not trivial, and there might be cases where projects from the same domain are not available.

The main problem of cross-project defect prediction is the heterogeneity of data: *It appears that every project has its own individual factors that make specific modules prone to defects and others unlikely to fail* [111]. Several approaches have been proposed in the literature to mitigate such a problem. Turban *et al.* [84] used nearest-neighbor filtering to fine tune cross-project defect prediction models. Unfortunately, such a filtering only reduces the gap between the accuracy of within- and cross-project defect prediction models. Cruz *et al.* [254] studied the application of a data transformation for building and using logistic regression models. They showed that simple log transformations can be useful when measures are not as spread as those measures used in the construction. Nam *et al.* [255] applied a data normalization (z-score normalization) for cross-project prediction in order to reduce data coming from different projects to the same interval. This data pre-processing is also used in Chapter 4 to reduce the heterogeneity of data. Turban *et al.* [256] analyzed the effectiveness of prediction models built on mixed data, i.e., within- and cross-project. Their results indicated that there are some benefits when considering mixed data. Nevertheless, the accuracy achieved considering project-specific data is greater than the accuracy obtained for cross-project prediction.

Menzies *et al.* [86] observed that prediction accuracy may not be generalizable within a project itself. Specifically, data from a project may be crowded in local regions which, when considered at a global level, may lead to different conclusions in terms of both quality

[2] A Radial Basis Function Network is a neural network with three layers: (i) the input layer which corresponds to the predictors, i.e., software metrics; (ii) the output layer which maps the outcomes to predict, i.e., the defect proneness of entities; (iii) the hidden layer used to connect the input layer with the output layer [252].

control and effort estimation. For this reason, they proposed a "local prediction" that could be applied to perform cross-project or within-project defect prediction. In the cross-project defect prediction scenario, let us suppose we have two projects, A and B, and suppose one wants to perform defect prediction in B based on data available for A. First, the approach proposed by Menzies *et al.* [86] clusters together similar classes (according to the set of identified predictors) into n clusters. Each cluster contains classes belonging to A and classes belonging to B. Then, for each cluster, classes belonging to A are used to train the prediction model, which is then used to predict defect-proneness of classes belonging to B. This means that n different prediction models are built. The results of an empirical evaluation indicated that conclusions derived from local models are typically superior and more insightful than those derived from global models. The approach proposed by Menzies *et al.* [86] is used in Chapter 4 as experimental baseline. A wider comparison of global and local models has been performed by Bettenburg *et al.* [87], who compared (i) local models, (ii) global models, and (iii) global models accounting for data specificity. Results of their study suggest that local models are valid only for specific subsets of data, whereas global models provide trends that are too general to be used in the practice.

All these studies suggested that cross-project prediction is particularly challenging and, due to the heterogeneity of projects, prediction accuracy might be poor in terms of precision, recall and F-score. Brahman *et al.* [88] argued that while broadly applicable, such measures are not well-suited for the quality-control settings in which defect prediction models are used. They show that, instead, the choice of prediction models should be based on both effectiveness (e.g., precision and recall) and inspection cost, which they approximate in terms of number of source code lines that need to be inspected to detect a given number/percentage of defects. By considering such factors, they found that the accuracy of cross-project defect prediction is adequate, and comparable to within-project prediction from a practical point of view.

Arisholm *et al.* [90] also suggested to measure the performance of prediction models in terms of cost and effectiveness for within-project prediction: a good model is the one that identifies defect-prone files with the best ratio between (i) effort spent to inspect such files, and (ii) number of defects identified. Once again, the cost was approximated by the percentage of lines of code to be inspected, while the effectiveness was measured as the percentage of defects found within the inspected code. However, they used classical predicting models (classification tree, PART, Logistic Regression, Back-propagation neural networks) to identify the defect-prone classes. Arisholm and Briand [89] used traditional logistic regression with a variety of metrics (history data, structural measures, etc.) to predict the defect-proneness of classes between subsequent versions of a Java legacy system, and used the cost-effectiveness to measure the performance of the obtained logistic models.

In previous studies, cost-effectiveness has been only used to assess the quality of a predictor, and they still relies on single-objective predictors (such as the logistic regression) which, by definition, find a model that minimizes the fitting error. With respect to all the works described above, this Chapter presents and evaluates a search-based approach to build multi-objective defect prediction models that take into account multiple criteria. Specifically,

for the first time we formulate the defect prediction problem as a multi-objective problem, then we apply search algorithms to produce a Pareto front of (near) optimal prediction models—instead of a single model as done in the past—with different effectiveness and cost values.

4.3 MODEP: Multi-Objective Defect Predictor

As every defect prediction approach, MODEP follows all the steps of the process described in Section 4.2. First, a set of predictors (e.g., product [235, 243] or process metrics [232]) is extracted for each class of a software project. The computed data is then preprocessed or normalized in order to reduce the effect of data heterogeneity. Such preprocessing step is particularly useful when performing cross-project defect prediction, as data from different projects (and in some case in the same project) have different properties [86]. Since a prediction model generally does not explicitly consider the local differences between different software projects when it tries to predict defects across projects, its performances can be unstable [86]. Once the data have been preprocessed, a machine learning technique is used to build a prediction model. In this chapter we evaluated two machine learning techniques, named logistic regression and decision trees, however other techniques can be applied without loss of generality.

4.3.1 Data Preprocessing

In cross-project defect prediction the software projects, that are used as training and test sets, are often heterogeneous because they exhibit different software metric distributions. For example, the average number of lines of code (LOC) of classes, which is a widely used (and obvious) defect predictor variable, can be quite different from a project to another. Hence, when evaluating predicting models on a software project with a software metric distribution that is different with respect to the data distribution used to build the models themselves, the predicting accuracy can be compromised [86]. In MODEP we perform a data *standardization*, i.e., we convert metrics into a z distribution, in order to reduce the effect of heterogeneity between different projects. Specifically, given the value of the metric m_i computed on class c_j of project P, denoted as $m_i(c_j, P)$, we convert it into:

$$\widetilde{m}_i(c_j, P) = \frac{m_i(c_j, P) - \mu(m_i, P)}{\sigma(m_i, P)}$$

In other words, we subtract from the value of the metric m_i the mean value $\mu(m_i, P)$ obtained across all classes of project P, and divide by the standard deviation $\sigma(m_i, P)$. Such a normalization transforms the data belonging to different projects to fall within the same range, by measuring the distance of a data point from the mean in terms of the standard deviation. The standardized data set has mean 0 and standard deviation 1, and it preserves the shape properties of the original data set (same skewness and kurtosis). A similar approach

was applied by Gyimothy *et al.* [243]. However, they used such preprocessing to reduce metrics to the same interval before combining them for within-project prediction, whereas we use it to reduce the effect of project heterogeneity in cross-project prediction. Recently, the usefulness of data normalization for cross-project prediction was also shown by Nam *et al.* [255], demonstrating how the predicting accuracy of a prediction model can be better when trained on normalized data.

4.3.2 Multi-Objective Defect Prediction Models

As explained in Section 4.2 a defect prediction model is a mathematical function/model $F : \mathbb{R}^n \rightarrow \mathbb{R}$, which takes as input a set of predictors and returns a scalar value that measures the likelihood that a specific software entity is defect-prone. Specifically, a model F combines the predictors into some classification/prediction rules through a set of scalar values $A = \{a_1, a_2, \ldots, a_k\}$ that is used to performs a specific combination. The number of scalar values and the type of classification/prediction rules depend on the model/function F itself. During the training process of the model F an optimization algorithm is used to find the set of values $A = \{a_1, a_2, \ldots, a_k\}$ that provides the best prediction of the outcome. For example, with linear regression techniques the predictors are combined through linear combination where the scalar values $A = \{a_1, a_2, \ldots, a_k\}$ are the linear combination coefficients. In this traditional formulation a solution is represented by only one (unique) set A of values, which minimizes only one objective function: the root-mean-square error (RMSE) [257]. Formally, the traditional defect prediction problem can be formulated as follows:

Problem 3. *Let $\{c_1, c_2, \ldots, c_n\}$ be the set of classes and let F be a specific machine learning model based on a set of combination coefficients $A = \{a_1, a_2, \ldots, a_k\}$. Find the set of coefficients A which minimizes root-mean-square error:*

$$\min RMSE = \sqrt{\sum_{i=1}^{n} \left(F_A(c_i) - DefectProne(c_i)\right)^2} \tag{4.1}$$

where $F_A(c_i)$ and $DefectProne(c_i)$ have value in $\{0; 1\}$ and represent the predicted defect-proneness and the actual defect-proneness of c_i.

Specifically, the objective function measures the number of classes that are erroneously classified within the training set: (i) defect-prone classes classified as non defect-prone (false negatives) and (ii) defect-free classes classified as defect-prone ones (false positives). However, focusing only on the prediction error might be not enough for building an effective and efficient prediction model. In fact, for the software engineer—who has to test/inspect the classes classified as defect-prone—the predicting error does not provide any insights on the effort required to analyze the identified defect-prone classes (that is a crucial aspect when prioritizing QA activities). Indeed, larger classes might require more effort to detect defects than smaller ones, because in the worst case the software engineer has to inspect the whole source code. Furthermore, it would be more useful to analyze early classes having a high

likelihood to be affected by more defects. Unfortunately, all these aspects are not explicitly captured by traditional single-objective formulation of the defect prediction problem.

For this reason, we suggest to move from this single-objective formulation of defect prediction towards a multi-objective one. The idea is to measure the *goodness* of a defect prediction models in terms of cost-effectiveness, that are by definition two contrasting goals. More precisely, we provide a new (multi-objective) formulation of the problem of finding defect prediction models:

Problem 4. *Given a set of classes* $\{c_1, c_2, \ldots, c_n\}$ *and given a specific machine learning model* F *based on a set of combination coefficients* $A = \{a_1, a_2, \ldots, a_k\}$*. Finding a set of values* $A = \{a_1, a_2, \ldots, a_k\}$ *that (near) optimizes the following objective functions:*

$$\begin{aligned} &\max \ \textit{effectiveness}(A) = \sum_{i=1}^{n} F_A(c_i) \cdot \textit{DefectProne}(c_i) \\ &\min \ \textit{cost}(A) = \sum_{i=1}^{n} F_A(c_i) \cdot LOC(c_i) \end{aligned} \tag{4.2}$$

where $F_A(c_i)$ *and* $\textit{DefectProne}(c_i)$ *have value in* $\{0, 1\}$ *and represent the predicted defect-proneness and the actual defect-proneness of* c_i*, respectively, while* $LOC(c_i)$ *measures the number of lines of code of* c_i*.*

In this formulation of the problem, we measure the effectiveness in terms of the number of actual defect-prone classes predicted as such. However, defect-prone classes could have different density of defects. In other words, there could be classes with only one defect and other classes with several defects. Thus, could be worthwhile—at the same cost—to focus the attention on classes having a high defect density. For this reason, we propose a second formulation of the problem:

Problem 5. *Given a set of classes* $\{c_1, c_2, \ldots, c_n\}$ *and given a specific machine learning model* F *based on a set of combination coefficients* $A = \{a_1, a_2, \ldots, a_k\}$*. Finding a set of values* $A = \{a_1, a_2, \ldots, a_k\}$ *that (near) optimizes the following objective functions:*

$$\begin{aligned} &\max \ \textit{effectiveness}(A) = \sum_{i=1}^{n} F_A(c_i) \cdot \textit{DefectNumber}(c_i) \\ &\min \ \textit{cost}(A) = \sum_{i=1}^{n} F_A(c_i) \cdot LOC(c_i) \end{aligned} \tag{4.3}$$

where $\textit{DefectNumber}(c_i)$ *denotes the actual number of defects in the class* c_i*.*

In both formulations, *cost* and *effectiveness* are two conflicting objectives, because one cannot increase the effectiveness (e.g., number of defect-prone classes correctly classified) without increasing (worsening) the inspection cost[3]. We do not to consider precision and

[3] In the ideal case the inspection cost is increased by the cost required to analyze the new classes classified as defect-prone.

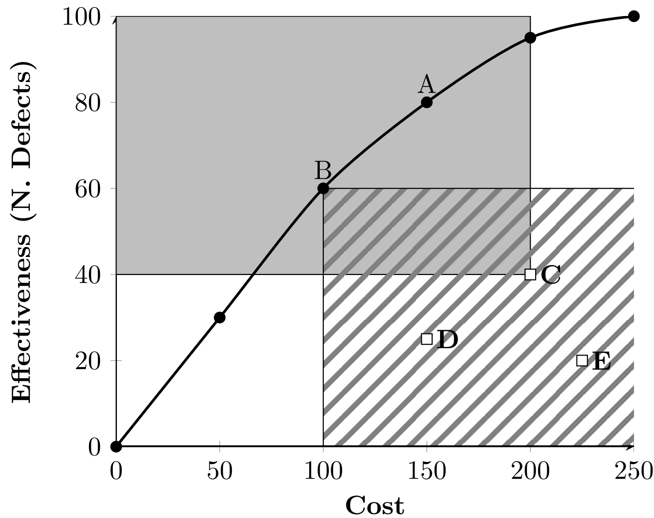


Figure 4.2: Graphical interpretation of Pareto dominance.

recall as the two contrasting objectives, because precision is less relevant than inspection cost—although proportional to it—when choosing the most suitable predictor.

Solving the multi-objective defect prediction problems defined above requires to find the set of solutions which represent optimal compromises between cost and effectiveness [135]. Hence, the goal becomes to find a multitude of optimal sets of decision coefficients A, i.e., a set of optimal predicting models. In the multi-objective paradigm, the concept of optimality is based on Pareto dominance relation between solutions [135]. Specifically, a solution (represented by a vector of values A_i) dominates another solution A_j (also written $A_i \geq_p A_j$) if and only if the values of two objective functions (effectiveness and cost) satisfy:

$$\text{cost}(A_i) \leq \text{cost}(A_j) \textit{ and } \text{effectiveness}(A_i) > \text{effectiveness}(A_j)$$
$$\textit{or}$$
$$\text{cost}(A_i) < \text{cost}(A_j) \textit{ and } \text{effectiveness}(A_i) \geq \text{effectiveness}(A_j)$$

Conceptually, the definition above indicates that A_i is better than A_j if and only if, at the same level of effectiveness, A_i has a lower inspection cost than A_j. Alternatively, A_i is better than A_j if and only if, at the same level of inspection cost, A_i has a greater effectiveness than A_j. Figure 4.2 provides a graphical interpretation of Pareto dominance in terms of effectiveness and inspection cost in the context of defect prediction. The solutions A and B highlighted in Figure 4.2 are Pareto-optimal since there is no other solution that provides a better effectiveness at the same cost and *vice-versa*. All solutions in the line-pattern rectangle (C, D, E) are dominated by B, because B is better in terms of both cost and effectiveness. Finally, all solutions in the gray rectangle (A and B) dominate solution C.

Among all the possible solutions, we are interested in finding all the solutions that are not

dominated by any other solution (*Pareto optimal set*) and the corresponding *objective vectors* (containing the values of two objective functions *effectiveness* and *cost*) which form the *Pareto frontier*. Identifying a Pareto frontier is particularly useful because the software engineer can use the frontier to make a well-informed decision that balances the trade-offs between the two objectives. In other words, the software engineer can choose the solution with lower inspection cost or higher effectiveness on the basis of the resources available for inspecting the predicted defect-prone classes. The two multi-objective formulations of the defect prediction problem can be applied to any predicting model, by identifying the Pareto optimal *decision vectors* that can be used to combine the predictors in classification/predicting rules. In this Chapter we provide a multi-objective formulation of two widely used models, i.e., logistic regression and decision trees. The details of both multi-objective logistic regression a multi-objective decision tree are reported in the following two subsections.

Multi-Objective Logistic Regression

One of the most used machine learning techniques for predicting the defect proneness of software entities is multivariate logistic regression, a generalization of the linear regression to binary classification, i.e. either a file is defect-prone or it is not in our case. For example, it was used by Zimmermann *et al.* [85] in the first work on cross-project defect prediction.

Let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of classes in the training set and let P be the corresponding *class-by-predictor matrix*, i.e., a $m \times n$ matrix, where m is the number of predictors and n is the number of classes in the training set, while its generic entry $p_{i,j}$ denotes the value of the ith predictor for the jth class. The mathematical function used for regression is called *logit*:

$$logit(c_j) = \frac{e^{\alpha+\beta_1\ p_{j,1}+\cdots+\beta_m\ p_{j,m}}}{1 + e^{\alpha+\beta_1\ p_{j,1}+\cdots+\beta_m\ p_{j,m}}} \tag{4.4}$$

where $logit(c_j)$ is the estimated probability that the jth class is defect-prone, while the scalars $(\alpha, \beta_1, \ldots, \beta_m)$ represent the linear combination coefficients for the predictors $p_{j,1}, \ldots, p_{j,m}$. The larger the absolute value of a coefficient β_i, the stronger the effect of the corresponding predictor p_i on the likelihood of a defect being detected in the entity c_j [235]. Using the *logit* function, it is possible to define a defect prediction model as follows:

$$F_A(c_j) = \begin{cases} 1 & \text{if } logit(c_j) > 0.5; \\ 0 & otherwise. \end{cases}$$

In the traditional single-objective formulation of the defect prediction problem, the predicted values $F_A(c_j)$ are compared with the actual defect-proneness of the classes in the training set, in order to find the set of decision scalars $(\alpha, \beta_1, \ldots, \beta_n)$ which minimizes the RMSE. Since the equation (4.4) cannot be solved analytically, the *maximum likelihood* [257] procedure is used to estimate the coefficients that minimize the prediction error, i.e. the difference between the predicted probability $P(c_j)$ and the observed outcome values. Maximum likelihood estimation is an iterative procedure that starts with arbitrary values of coefficients for

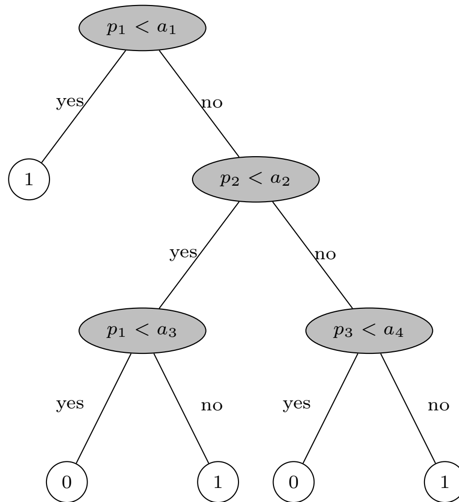


Figure 4.3: Decision tree for defect prediction.

a set of predictors and determines the direction and size of change in the coefficients that will maximize the likelihood of obtaining the observed values. The residuals for the predictive model based on those coefficients are then tested and another determination of direction and size of change in coefficients is made. This procedure repeats until each coefficient converges to a steady value. Clearly, such a method balances between the capability of the approach to identify defect-prone components (true positive), and the limited number of false positives.

This single-objective model can be converted in a *multi-objective logistic regression model* by using the same *logit* function, but searching for multiple sets of Pareto optimal coefficients which optimizes (i) the inspection cost, and (ii) the effectiveness of the prediction (that can be the number of defective classes or defect density). In this way, we can yield a set of logistic models that provide (near) optimal compromises between the two contrasting objectives. In particular, given a solution $A = (\alpha, \beta_1, \ldots, \beta_m)$ and the corresponding prediction values $F_A(c_j)$ computed applying the equation 4.4, we can evaluate the Pareto optimality of A using the *cost* and *effectiveness* functions (according to the equations 4.2 or the equations 4.3). Once obtained the set of Pareto optimal defect prediction models, the software engineer can select the one that she considers the most appropriate and uses it for predicting the defect-proneness of other classes.

Multi-Objective Decision Trees

Decision trees are also widely used for defect prediction and they are generated according to a specific set of classification rules [258, 85]. A decision tree has, clearly, a tree structure where the leaf nodes are the predicting outcomes (class defect-proneness in our case), while the other nodes, often called as *decision nodes*, contain the decision rules. Each decision rule is based on a predictor p_i, and it partitions the decision in two branches according to a

specific *decision coefficient* a_i. In other words, a decision tree can be viewed as a sequence of questions, where each question depends on the previous questions. Therefore, a decision corresponds to a specific path on the tree. Figure 4.3 shows a typical example of decision tree used for defect prediction. Each decision node has the form *if* $p_i < a_i$, while each leaf node contains 1 (defective class) or 0 (non defective class). When a given instance has to be classified, the tree is traversed from the root node to bottom until a leaf node is reached. The decision about which branch to follow is performed at each node of the tree on the basis of the test condition on the predictor p_i corresponding to that node. Finally, each leaf node is a linear regression model associated to obtain the predicting outcome. Hence, the classification of a given instance is performed by following all paths for which all decision nodes are true and summing the predicting values that are traversed along the corresponding path.

The process of building a decision tree consists of two main steps: (i) generating the structure of the tree, and (ii) generating the decision rules for each decision node according to a given set of decision coefficients $A = \{a_1, a_2, \ldots, a_k\}$. Several algorithms can be used to build the structure of a decision tree [259] that can use a bottom-up or a top-down approach. In this Chapter we use the **ID3** algorithm developed by Quinlan [260], which applies a top-down strategy with a greedy search through the search space to derive the best structure of the tree. In particular, starting from the root node, the **ID3** algorithm uses the concepts of *Information Entropy* and *Information Gain* to assign a given predictor p_i to the current node[4], and then to split each node in two children, partitioning the data in two subsets containing instances with similar predictors values. The process continues iteratively until no any further split affects the *Information Entropy.*

Once the structure of the tree is built, the problem of finding the best decision tree in the traditional single-objective paradigm consists of finding for the all decision nodes the set of coefficients $A = \{a_1, a_2, \ldots, a_k\}$ which minimizes the root square prediction error. Similarly to the logistic regression, we can move from the single objective formulation towards a multi-objective one by using the two-objective functions reported in equations 4.2 or 4.3. Specifically, we propose to find a multiple sets of decision coefficients $A = \{a_1, a_2, \ldots, a_n\}$ that (near) represent optimal compromises between (i) the inspection cost, and (ii) the prediction effectiveness. Note that MODEP only acts on the decision coefficients, while it uses a well-known algorithm, i.e., the **ID3** algorithm, for building the tree structure. This means that the set of decision trees on the Pareto front have the same structure but different decision coefficients.

4.3.3 Training the Multi-Objective Predictors using Genetic Algorithms

The problem of determining the coefficients for the logistic model or for the decision tree can be seen as an optimization problem with two conflicting *objective functions.* In MODEP we decided to solve such a problem using a multi-objective GA. The first step for the definition

[4] A predictor p_i is assigned to a given node n_j if and only if the predictor p_i is the larger informational gain with respect to the other predictors.

of a GA is the solution representation. In MODEP, a solution (chromosome) is represented by a vector of values. For the logistic regression, the chromosome contains the coefficients of the *logit* function. Instead, the decision coefficients of the decision nodes are encoded in the chromosome in the case of decision trees. For example, a chromosome for the decision tree reported in Figure 4.3 is $A = \{a_1, a_2, a_3, a_4\}$ which is the set of decision coefficients used to make a decision on each decision node.

Once the model coefficients are encoded as chromosomes, the multi-objective genetic algorithm is used to determine them. Among all the variants of MOGAs that have been proposed in literature, we used NSGA-II [107] that is one of the most popular. A detailed description of NSGA-II and its behavior (as well as its pseudo-code) is described in Section 2.6. It is worth noting that we apply NSGA-II for defining decision/coefficient values on the training set, obtaining a set of non-dominated solutions that provide different effectiveness and cost values (on the classes in the training set). After that, each Pareto-optimal solution can be used to build a model (based on logistic regression or decision tree) for predicting defect-prone classes on other classes.

4.4 Design of the Empirical Study

This section describes the study we conducted to evaluate the proposed multi-objective formulation of the defect prediction problem. The description follows a template originating from the Goal-Question-Metric paradigm [225].

4.4.1 Definition and Context

The *goal* of the study is to evaluate MODEP, with the *purpose* of investigating the benefits introduced by the proposed multi-objective prediction in a cross-project context. The reason why we focus on cross-project prediction is because (i) this is very useful when project history data is missing and challenging at the same time [85]; and (ii) as pointed out by Brahman *et al.* [88], cross-project prediction may turn out to be cost-effective while not exhibiting high precision values.

The *quality focus* of the study is the capability of the proposed approach to highlight likely defect-prone classes in a cost-effective way, i.e., recommending the QA team to perform a cost-effective inspection of classes giving much priority to classes that have a higher defect density and in general to maximize the number of defects identified for a given cost. The *perspective* is of researchers aiming at developing a better, cost-effective defect prediction model, also able to work well for cross-project prediction, where the availability of project data does not allow a reliable within-project defect prediction.

The *context* of our study consists of 10 Java projects where information on defects is available. All of them come from the Promise repository[5]. A summary of the project characteristics is reported in Table 4.1. All the datasets report the actual defect-proneness

[5]https://code.google.com/p/promisedata/

Table 4.1: Java projects used in the study.

Name	Release	Classes	Defect-prone classes	(%)
Ant	1.7	745	166	22%
Camel	1.6	965	188	19%
Ivy	2.0	352	40	11%
Jedit	4.0	306	75	25%
Log4j	1.2	205	189	92%
Lucene	2.4	340	203	60%
Poi	3.0	442	281	64%
Prop	6.0	661	66	10%
Tomcat	6.0	858	77	9%
Xalan	2.7	910	898	99%

of classes, plus a pool of metrics used as predictors, i.e., LOC and the Chidamber & Kemerer Metric suite [231], namely Weighted Method Count (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for Classes (RFC), and Lack of Cohesion of Methods (LCOM).

4.4.2 Research Questions

In the context of our study we formulated the following research questions:

RQ$_1$ *How does MODEP perform compared to single-objective prediction?* This research question aims at evaluating, from a *quantitative* point of view, the benefits introduced by the multi-objective definition of a cross-project defect prediction problem. We will evaluate multi-objective predictors based on logistic regression and decision trees. Also, we consider models producing Pareto fronts of predictors (i) between LOC and number of predicted defect-prone classes, and (ii) between LOC and number of predicted defects.

RQ$_2$ *How does MODEP perform compared to the local prediction approach?* This research question aims at comparing the cross-project prediction capabilities of MODEP with those of the approach—proposed by Menzies *et al.* [86]—that uses local prediction to mitigate the heterogeneity of projects in the context of cross-project defect prediction. We consider such an approach as a baseline for comparison because it is considered as the state-of-the-art for cross-project defect prediction.

In addition, we provide *qualitative* insights about the practical usefulness of having Pareto fronts of defect predictors instead of a single predictor. Also, we will highlight how, for a given machine learning model (e.g., logistic regression or decision trees) one can choose the appropriate rules (on the various metrics) that leads towards a prediction achieving a given level of cost-effectiveness.

4.4.3 Variable Selection

Our evaluation studied the effect of the following independent variables:

- *Machine learning algorithm:* both single and multi-objective prediction techniques are implemented using logistic regression and decision trees. We used the logistic regression and the decision tree implementations available in *MATLAB* [261]. The *glmfit* routine was used for training the logistic regression model with *binomial distribution* and using the *logit* function as generalized linear model, while for the decision tree model we used the *classregtree* class to built decision trees.

- *Objectives (single* vs. *multi-objective prediction):* the main goal of $\mathbf{RQ}_1$ is to compare single-objective models with multi-objective predictors. The former is a traditional machine learning technique in which the model is built by fitting data in the training set. The latter, as explained in Section 4.3.2, are a set of Pareto-optimal predictors built by a multi-objective GA, achieving different cost-effectiveness tradeoffs.

- *Training (within-project* vs. *cross-project prediction)*: we compare the prediction capability in the context of within-project prediction with those of cross-project prediction. The conjecture we want to test is wether the cross-project strategy is comparable/better than the within-project strategy in terms of cost-effectiveness when using MODEP.

- *Prediction (local* vs. *global):* we consider—within $\mathbf{RQ}_2$–both local prediction (using the clustering approach by Menzies *et al.* [86]) and global prediction.

In terms of dependent variables, we evaluated our models using (i) code *inspection cost*, measured as the KLOC of the of classes predicted as defect-prone (as done by Brahman *at al.* [88]), and (ii) recall, which provides a measure of the model effectiveness. In particular, we considered two granularity levels of recall by counting (i) the number of defect-prone classes correctly classified; and (ii) the number of defects:

$$recall_{class} = \frac{\sum_{i=1}^{n} F_A(c_i) \cdot \text{DefectProne}(c_i)}{\sum_{i=1}^{n} \text{DefectProne}(c_i)}$$

$$recall_{defect} = \frac{\sum_{i=1}^{n} F_A(c_i) \cdot \text{DefectNumber}(c_i)}{\sum_{i=1}^{n} \text{DefectNumber}(c_i)}$$

where $F_A(c_i)$ denotes the predicted defect proneness of the class c_i, DefectProne(c_i) measures its actual defect proneness and DefectNumber(c_i) is the number of defects in c_i. Hence, the *recall* metric computed at *class granularity* level corresponds to the traditional *recall* metric

which measures the percentage of defect prone classes that are correctly classified. The recall metric computed at *defect granularity* provides a weighted version of recall where the weights are represented by the number of defects in the classes. The idea of this metric is that at the same level of inspection cost, it would be more effective to inspect early classes having a higher defect density, i.e., classes, that are affected by a higher number of defects.

During the analysis of the results, we also report precision to facilitate the comparison with other models:

$$precision = \frac{\sum_{i=1}^{n} F_A(c_i) \cdot \text{DefectProne}(c_i)}{\sum_{i=1}^{n} F(c_i)}$$

It is important to note that precision, recall and cost were reported according to the prediction of defect-prone classes. It is advisable not to aggregate them with the prediction of non-defect-prone classes: a model with high (overall) precision (say 90%) when the number of defect-prone classes is very limited (say 5%), performs worse than a constant classifier. Similar considerations can be done considering the cost instead of the precision: a model with lower cost (lower number of KLOC to analyze), but with a limited number of defect-prone classes might not be effective.

We use a cross-validation procedure [262] to compute precision, recall and inspection cost. Specifically, for the within-project prediction, we used a 10-fold cross validation implemented in MATLAB by the *crossvalind* routine, which randomly partitions a software project into 10 equal size folds; we used 9 folds as training set and the 10th as test set. This procedure was performed 10 times, with each of the fold used exactly once as the validation data. For the cross-project prediction, we applied a similar procedure, removing each time a project from the set, training on 9 projects and predicting on the 10th one.

4.4.4 Analysis Method

To address $\mathbf{RQ}_1$, we compare the performance of MODEP with those of the single objective (i) within project (ii) and cross-project predictors. For both MODEP and the single objective predictors, we report the performance of logistic regression and decision trees. Also, we analyze the two multi-objective models separately, i.e., the one that considers as objectives cost and defect-prone classes, and the one that consider cost and number of defect. In order to compare the experimented predictors, we first visually compare the Pareto fronts (more specifically, the line obtained by computing the median over the Pareto fronts of 30 GA runs) and the performance (a single dot in the cost-effectiveness plane) of the single-objective predictors. Then, we compare the *recall* and *precision* of MODEP and single-objective predictors at the same level of inspection cost. Finally, by using the precision and recall values over the 10 projects, we also statistically compare MODEP with the single objective models using the two-tailed Wilcoxon paired test [263] to determine whether the following null hypotheses could be rejected:

- H_{0_R}: *there is no significant difference between the recall of MODEP and the recall of the single-objective predictor.*

- H_{0_P}: *there is no significant difference between the precision of MODEP and the precision of the single-objective predictor.*

Note that the comparison is made by considering both MODEP and the single-objective predictor implemented with logistic regression and decision tree, and with the two different kinds of recall measures (based on the proportion of defect-prone classes and of defects). In addition, we use the Wilcoxon test, because it is non-parametric and does not require any assumption upon the underlying data distribution; also, we perform a two-tailed test because we do not know a priori whether the difference is in favor of MODEP or of the single-objective models. For all tests we assume a significance level $\alpha = 0.05$, i.e., 5% of probability of rejecting the null hypothesis when it should not be rejected.

To address $\mathbf{RQ}_2$, we compare MODEP with the local prediction approach proposed by Menzies *et al.* [86]. In the following, we briefly explain our re-implemented of the local prediction approach using the *MATLAB* environment [261] and, specifically, the *RWeka* and *cluster* packages. The prediction process consists of three steps:

1. *Data preprocessing*: we preprocess the data set as described in Section 4.3.1.

2. *Data clustering*: we cluster together classes having similar characteristics (corresponding to the WHERE heuristic by Menzies *et al.* [86]). We use a traditional multidimensional scaling algorithm (MDS)[6] to cluster the data and using the Euclidean distance to compute the dissimilarities between classes. We use its implementation available in MATLAB with the *mdscale* routine and using as number of iterations $it = \sqrt{n}$, where n is the number of classes in the training set (such a parameter is the same used by Menzies *et al.* [86]). As demonstrated by Yang *et al.* in [264] such an algorithm is exactly equivalent to the *FASTMAP* algorithm used by Menzies *et al.* [86], except for the computation cost. FASTMAP approximates the classical MDS by solving the problem for a subset of the data set, and by fitting the remainder solutions [264]. A critical factor in the local prediction approach is represented by the number of clusters to be considered. To this aim, we used a widely-adopted approach from the cluster literature, i.e., the Silhouette coefficient [223]. The Silhouette coefficient is computed for each class to be clustered using the concept of cluster centroid[7]. Based on this approach, in our study we found a number of clusters k =*10* to be optimal.

3. *Local prediction*: finally, we perform a local prediction within each cluster identified using MDS. Basically, for each cluster obtained in the previous step, we use classes from $n-1$ projects to train the model, and then we predict defects for classes of the

[6]Specifically we used a metric based multidimensional scaling algorithm, where the metric used is the Euclidean distance in the space of predictors.

[7]The definition of Silhouette Coefficient is described in previous chapters. Thus, in this chapter we do not re-propose it. For the definition we remind Section 3.3.

remaining project. We use an association rule learner to generate a predicting model according to the cluster-based cross-project strategy (WHICH heuristic by Menzies *et al.* [86]). We used a MATLAB's tool, called ARMADA[8], which provides a set of routines for generating and managing association rule discovery.

4.4.5 Implementation and Settings of the Genetic Algorithm

MODEP has been implemented using *MATLAB Global Optimization Toolbox* (release R2011b). In particular, the *gamultiobj* routine was used to run the NSGA-II algorithm, while the routine *gaoptimset* was used to set the GA parameters. We used the GA configuration typically used for numerical problems [135]:

- **Population size:** we choose a moderate population size with $p = 200$.
- **Initial population:** for each software system the initial population is uniformly and randomly generated within the solutions space. Since such a problem in unconstrained, i.e., there are no upper and lower bounds for the values that the coefficients can assume, the initial population was randomly and uniformly generated in the interval $[-10; 10]^n$, where n is the length of the chromosomes.
- **Number of generations:** we set the maximum number of generation equal to 400.
- **Selection**: we used the tournament selection operator, with tournament size $k = 4$, as default for the MATLAB implementation of NSGA-II.
- **Crossover operator**: we use the arithmetic crossover [261] with probability $P_c = 0.60$, which is one of the most used crossover operator for real-coded problem.
- **Mutation function:** we use a uniform mutation function with probability $p_m = 1/n$ where n is the size of the chromosomes (solutions representation).
- **Stopping criterion:** if the average Pareto spread is lower than 10^{-8} in the subsequent 50 generations, then the execution of NSGA-II is stopped. The average Pareto spread measures the average distance between individuals of two Pareto fronts obtained by two subsequent generations. Thus, the average Pareto spread is used to measure whether the obtained Pareto front does not change across generations (i.e., whether NSGA-II converged).

The algorithm has been executed 30 times on each object program to account the inherent randomness of GAs [265]. Then, we select a Pareto front composed of points achieving the median performance across the 30 runs.

[8]http://www.mathworks.com/matlabcentral/fileexchange/3016-armada-data-mining-tool-version-1-4

4.5 Study Results

This section discusses the results of our study aimed at answering the research questions formulated in Section 4.4.2.

4.5.1 RQ_1: *How does MODEP perform compared to single-objective prediction?*

In this section we compare MODEP with a single-objective defect predictor. We first discuss the results achieved when using as objective functions inspection cost and number of defect-prone classes classified as such. Then, we discuss the results achieved when considering as objectives inspection cost and number of defects (over the total number of defects) contained in the defect-prone classes classified as such.

MODEP based on inspection cost and defect-prone classes.

Figures 4.4 and 4.5 show the Pareto fronts achieved by MODEP (using logistic regression and decision trees), when predicting *defect-prone classes*—and optimizing the inspection cost and the number of defect-prone classes identified—on each of the 10 projects after training the model on the remaining 9. The plots also show, as single points: (i) the single-objective *cross-project* logistic regression and decision tree predictors (as a black triangle for logistic regression, and gray triangle for decision trees); and (ii) the single-objective *within-project* for logistic and decision tree predictors (a black square for the logistic and a gray square for the decision tree), where the prediction has been performed using 10-fold cross-validation within the same project.

A preliminary analysis indicates that, generally, the solutions (sets of predictors) provided by MODEP (based on logistic regression or decision tree) dominate the solutions provided by both cross- and within-project single objective models. This means that the Pareto optimal predictors provided by MODEP are able to predict a larger number of defect-prone classes with a lower inspection cost. Only in two cases MODEP is not able to overcome the single-objective predictors. Specifically, for *Tomcat* the single-objective solutions (both cross- and within- project) are quite close to the Pareto fronts provided by MODEP based on decision trees, while for *Camel* the within-project decision tree dominates the solutions obtained by MODEP.

In order to provide a deeper comparison of the different prediction models, Table 4.2 compares the performances of MODEP with those of the cross project single-objective predictors (both logistic and decision tree predictors) in terms of *precision* and $recall_{class}$. Specifically, the table reports the $recall_{class}$ and the precision of the two models for the same level of inspection cost. Results indicate that the logistic model MODEP always achieves better $recall_{class}$ levels. In particular, for 5 systems (*Ant*, *Log4j*, *Lucene*, *Poi*, and *Prop*) the $recall_{class}$ is greater (of at least 10%) than the $recall_{class}$ of the single-objective predictors. However, the precision generally decreases, even if in several cases the decrement is negligible.

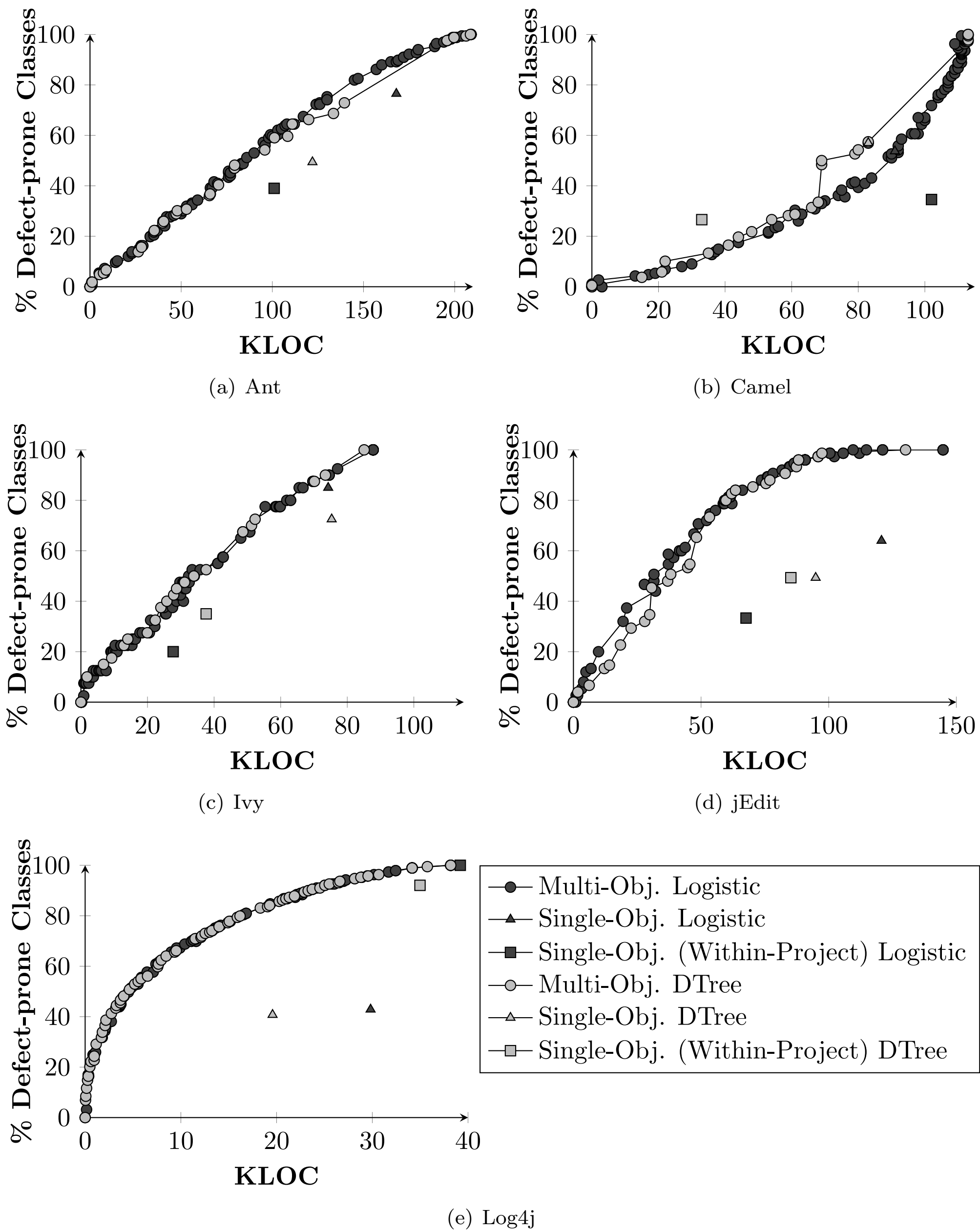


Figure 4.4: Performances of predicting models achieved when optimizing inspection cost and number of defect-prone classes.

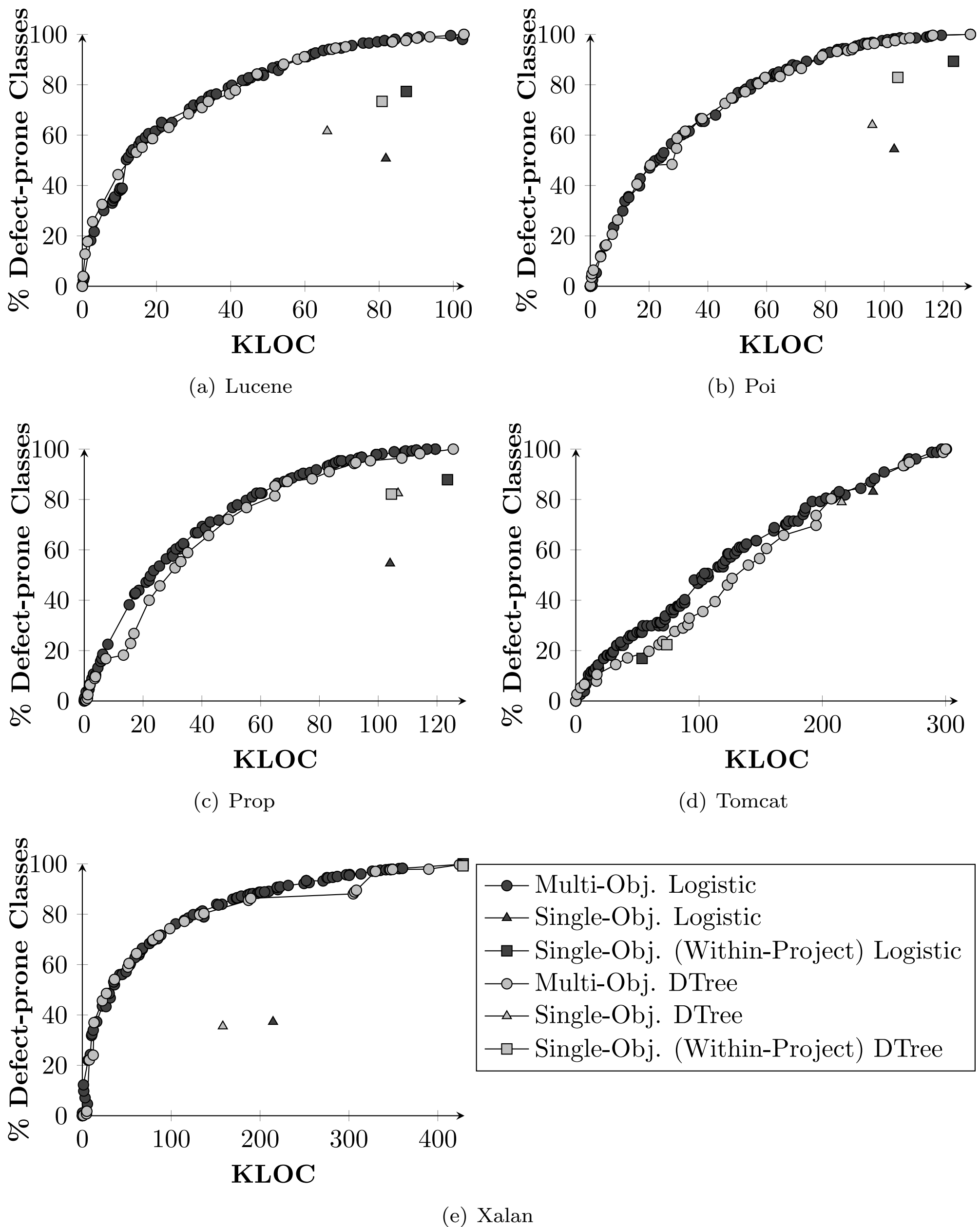


Figure 4.5: Performances of predicting models achieved when optimizing inspection cost and number of defect-prone classes.

Table 4.2: MODEP vs. single-objective predictors when optimizing inspection cost and number of defect-prone classes identified. Single-objective predictors are also applied using a within-project training strategy.

System	Metric	Logistic			Dtree			Logistic			Dtree		
		SO-C	**MO**	**Diff**	**SO-C**	**MO**	**Diff**	**SO-W**	**MO**	**Diff**	**SO-W**	**MO**	**Diff**
Ant	Cost	167	167	-	121	121	-	101	101	-	104	104	-
	$Recall_{class}$	0.77	0.90	+0.13	0.49	0.68	+0.19	0.39	0.60	+0.21	0.43	0.60	+0.17
	precision	0.43	0.21	-0.22	0.50	0.17	-0.33	0.68	0.52	-0.16	0.35	0.15	-0.20
Camel	Cost	93	93	-	83	83	-	13	13	-	33	33	-
	$Recall_{class}$	0.54	0.59	+0.05	0.57	0.57	-	0.09	0.09	-	0.25	0.25	-
	precision	0.26	0.27	+0.01	0.37	0.37	-	0.54	0.54	-	0.33	0.33	-
Ivy	Cost	74	74	-	75	75	-	28	28	-	38	38	-
	$Recall_{class}$	0.83	0.90	+0.07	0.72	0.90	+0.18	0.25	0.40	+0.15	0.35	0.52	+0.17
	precision	0.27	0.11	-0.16	0.22	0.21	-0.01	0.50	0.06	-0.44	0.37	0.07	-0.30
jEdit	Cost	121	121	-	95	95	-	66	66	-	85	85	-
	$Recall_{class}$	0.64	1.00	-	0.49	0.97	+0.38	0.33	0.84	+0.51	0.49	0.91	+0.42
	precision	0.42	0.25	-0.19	0.66	0.24	-0.42	0.66	0.22	-0.44	0.50	0.23	-0.27
Log4j	Cost	30	30	-	20	20	-	38	38	-	35	35	-
	$Recall_{class}$	0.42	0.96	+0.54	0.41	0.85	+0.44	0.99	0.99	-	0.92	0.99	+0.07
	precision	0.94	0.92	-0.02	0.93	0.92	-0.01	0.92	0.92	-	0.93	0.92	-0.01
Lucene	Cost	83	83	-	66	66	-	86	86	-	81	81	-
	$Recall_{class}$	0.53	0.97	+0.44	0.62	0.94	32	0.77	0.99	+0.22	0.73	0.95	+0.13
	precision	0.80	0.59	-0.21	0.63	0.59	-0.04	0.74	0.60	-0.14	0.74	0.59	-0.15
Poi	Cost	102	102	-	96	96	-	120	120	-	104	104	-
	$Recall_{class}$	0.53	0.98	+0.45	0.64	0.96	+0.32	0.90	1.00	+0.10	0.83	0.96	+0.13
	precision	0.87	0.63	-0.24	0.73	0.63	-0.10	0.79	0.64	-0.15	0.81	0.64	-0.23
Prop	Cost	76	76	-	107	107	-	74	74	-	104	104	-
	$Recall_{class}$	0.69	0.91	+0.22	0.83	0.97	+0.14	0.87	0.90	+0.03	0.82	0.95	+0.13
	precision	0.67	0.62	-0.05	0.81	0.63	-0.18	0.77	0.61	-0.16	0.83	0.63	-0.20
Tomcat	Cost	214	214	-	241	241	-	64	64	-	54	54	-
	$Recall_{class}$	0.82	0.82	-	0.84	0.86	+0.02	0.18	0.29	+0.11	0.18	0.18	-
	precision	0.21	0.08	-0.13	0.22	0.08	-0.14	0.58	0.04	-0.53	0.59	0.59	-
Xalan	Cost	285	285	-	158	158	-	429	429	-	428	428	-
	$Recall_{class}$	0.38	0.95	-	0.36	0.81	+0.45 6	1.00	1.00	-	0.99	1.00	+0.01
	precision	1	0.99	-0.01	0.99	0.98	-0.01	0.99	0.99	-	0.99	0.99	-

SO-C: Single-Objective Cross-project; **SO-W**: Single-Objective Within-project; **MO**: Multi-Objective

The same considerations also apply when MODEP uses decision trees.

We also compare MODEP with the single-objective predictors trained using a within-project strategy. This analysis is necessary to analyze to what extent MODEP trained with a cross-project strategy is comparable/better than single-objective predictors trained with a within-project strategy in terms of cost-effectiveness. Table 4.2 reports the achieved results in terms of *precision* and $recall_{class}$ for the same level of inspection cost. Not surprisingly, the within-project logistic predictor achieves, for 7 out of 10 projects (*Ant*, *Ivy*, *jEdit*, *Lucene*, *Poi*, *Prop*, and *Tomcat*), a better *precision*. Instead, the *precision* is the same for both logistic-based models for *Log4j*, *Camel*, and *Xalan*. Similarly, the within-project decision tree predictor achieves, for 9 out of 10 projects, a better *precision*. However, the difference between the *precision* values achieved by MODEP and the single-objective predictor are lower then 5% on 4 projects. Thus, the within-project single-project predictors achieve—for both logistic and decision tree—better *precision* than MODEP trained with a cross-project strategy. These results are consistent with those of a previous study [85], and show that, in general, within-project prediction outperforms cross-project prediction (and when this does not happen, performances are very similar). However, although the *precision* decreases,

MODEP is able to generally achieve higher $recall_{class}$ values using both the machine learning algorithms. This means that—for the same cost (KLOC)—the software engineer has to analyze more false positives, but she is also able to identify more defect-prone classes.

All these findings are also confirmed by our statistical analysis. As for the logistic regression, the Wilcoxon test indicate that the precision of MODEP is significantly lower (p-value = 0.02) than for the single-objective, but at the same time the $recall_{class}$ is also significantly greater (p-value < 0.01) than for the single-objective. Similarly, for the decision tree we also obtained a significant difference in terms of both precision (it is significantly lower using MODEP, with p-value < 0.01) and $recall_{class}$ (it is significantly higher using MODEP with p-value < 0.01). Thus, we can reject both the null hypotheses, H_{0_R} in favor of MODEP and H_{0_P} in favor of the single-objective predictors. This means that, for the same inspection cost, MODEP achieves a lower prediction precision, but it increases the number of defect-prone classes identified, with respect to the single-objective predictors.

MODEP based on inspection cost and number of defects.

Figure 4.6 and 4.6 show the Pareto fronts produced by MODEP when optimizing inspection cost and number of defects. Also in this case, the models were evaluated on each of the 10 projects after training the model on the remaining 9.

A preliminary analysis shows that, also when considering the number of defects as effectiveness measure, MODEP dominates the solutions provided by both cross- and within-project single objective models, using either logistic regression or decision trees. This means that the Pareto-optimal predictors provided by MODEP are able to predict classes having more defects with a lower inspection cost. Only in few cases, i.e., for *Tomcat* and *Ivy*, the single-objective solutions (both cross- and within-project) are quite close to the Pareto fronts provided by MODEP while only in one case, i.e., for *Ant*, the within-project logistic regression dominates the solutions achieved by MODEP.

As done for the previous two-objective formulation of the defect prediction problem, Table 4.3 reports the performances, in terms of *precision* and $recall_{defect}$ achieved by the experimented defect prediction models for the same inspection costs. Results indicate that MODEP is able to provide better performance than the corresponding single-objective predictors in terms of number of predicted defects—i.e., contained in the classes identified as defect-prone—at the same cost. Specifically, in 8 out of 10 projects, and for both logistic regression and decision trees, the number of defects contained in the classes predicted by MODEP is greater than for defect-classes predicted by the single-objective predictors. The only exceptions are represented by *Tomcat* for the logistic regression and by *Camel* for the decision trees, where the number of defects is the same for both MODEP and single-objective cross-project predictors. In summary, also when formulating the multi-objective problem in terms of cost and number of defects, MODEP is able to increase the number of defects in the predicted classes. However, the prediction achieved with single-objective predictors provides a higher precision. We also compare MODEP with single-objective defect predictors trained using a within-project strategy (see Table 4.3). Results indicate that MODEP is able to

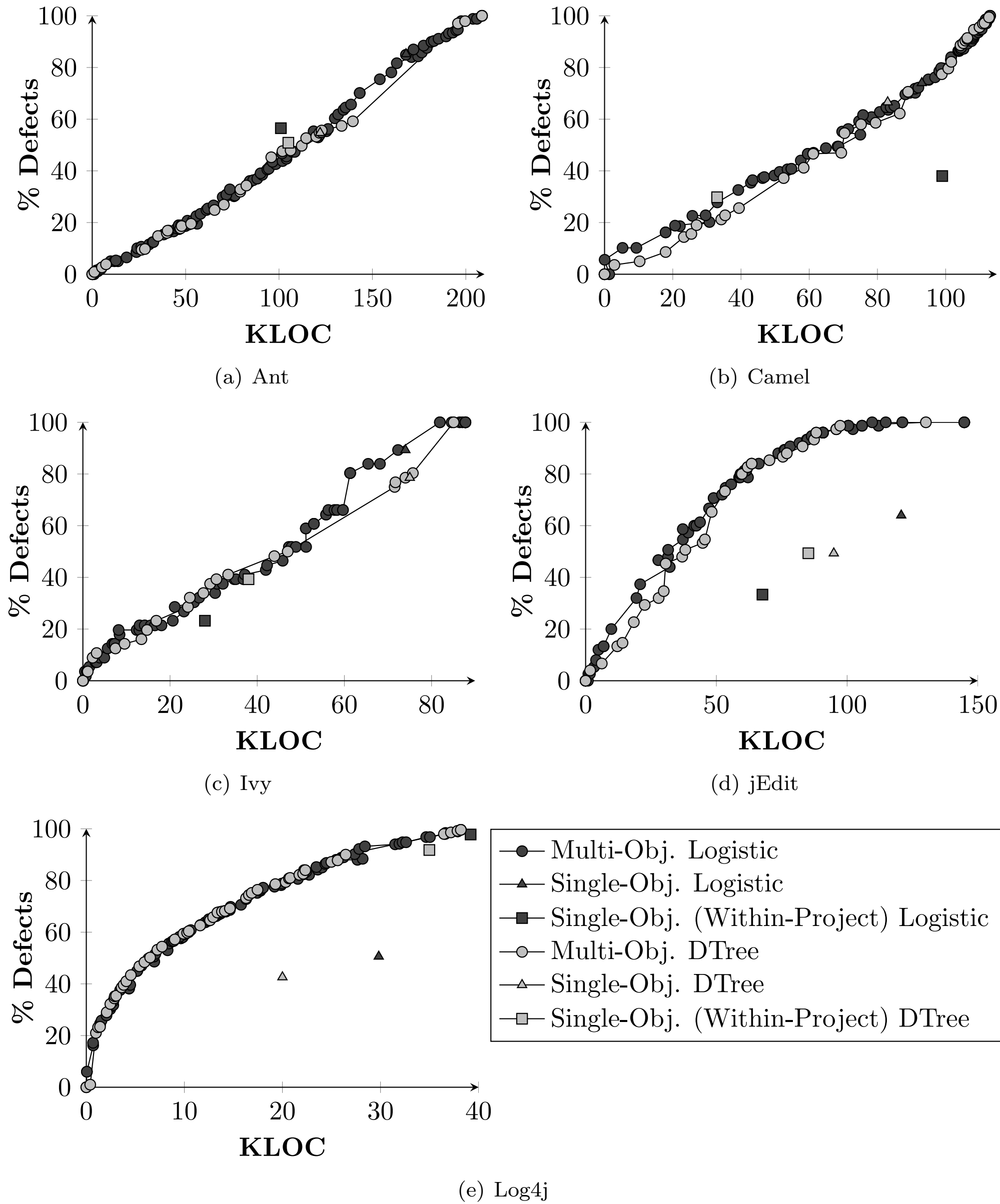


Figure 4.6: Performances of predicting models achieved when optimizing inspection cost and number of defects.

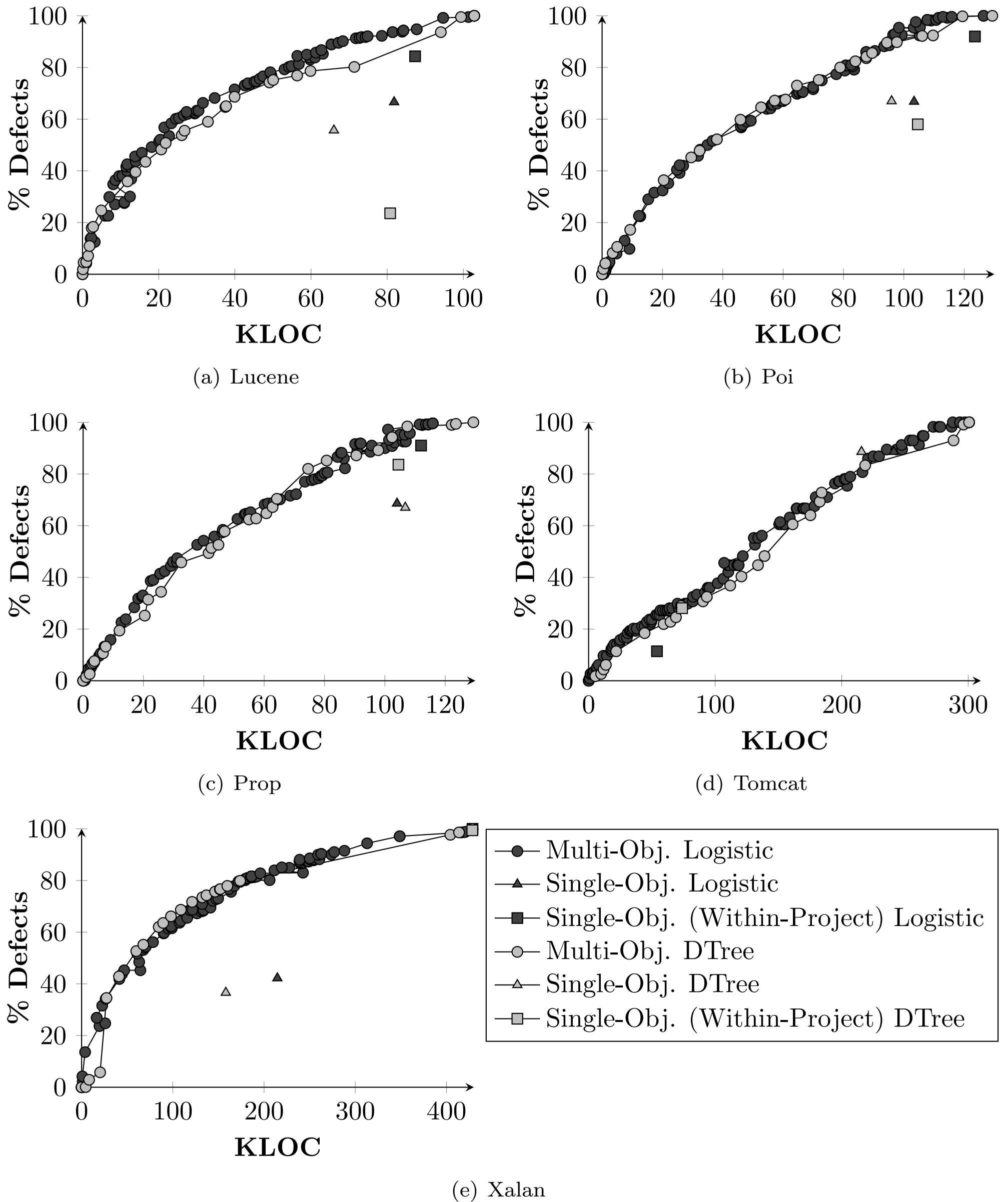


Figure 4.7: Performances of predicting models achieved when optimizing inspection cost and number of defects.

Table 4.3: MODEP vs. single-objective predictors when optimizing inspection cost and number of defects identified. Single-objective predictors are also applied using a within-project training strategy.

System	Metric	Logistic			Dtree			Logistic			Dtree		
		SO-C	**MO**	**Diff**	**SO-C**	**MO**	**Diff**	**SO-W**	**MO**	**Diff**	**SO-W**	**MO**	**Diff**
Ant	Cost	167	168	-	121	121	-	101	101	-	104	104	-
	$recall_{defects}$	0.85	0.85	-	0.64	0.64	-	0.57	0.48	-0.09	0.51	0.49	-0.03
	precision	0.43	0.43	-	0.50	0.56	+0.06	0.68	0.25	-0.43	0.35	0.30	-0.05
Camel	Cost	93	92	-1	83	83	-	13	13	-	33	33	-
	$recall_{defects}$	0.74	0.75	+0.01	0.67	0.62	-0.05	0.15	0.15	-	0.30	0.30	-
	precision	0.26	0.26	-	0.37	0.28	-0.09	0.54	0.50	-0.04	0.33	0.32	-0.01
Ivy	Cost	74	74	-	75	75	-	28	28	-	38	38	-
	$recall_{defects}$	0.89	0.93	+0.04	0.79	0.80	0.01	0.29	0.30	+0.01	0.39	0.45	+0.06
	precision	0.27	0.11	-0.16	0.22	0.16	-0.06	0.71	0.45	-0.26	0.37	0.35	-0.02
jEdit	Cost	121	121	-	95	95	-	66	66	-	85	85	-
	$recall_{defects}$	0.64	1.00	+0.36	0.49	0.97	+0.48	0.33	0.84	+0.51	0.49	0.93	+0.44
	precision	0.42	0.24	-0.20	0.66	0.24	-0.42	0.66	0.22	-0.44	0.50	0.23	-0.27
Log4j	Cost	30	30	-	20	20	-	38	38	-	35	35	-
	$recall_{defects}$	0.51	0.90	+0.39	0.42	0.79	+0.37	0.98	1.00	+0.02	0.92	0.98	+0.06
	precision	0.94	0.93	-0.01	0.93	0.92	-0.01	0.92	0.92	-	0.93	0.92	0.01
Lucene	Cost	83	83	-	66	66	-	86	86	-	81	81	-
	$recall_{defects}$	0.67	0.94	+0.27	0.56	0.79	+0.23	0.84	0.95	+0.11	0.24	0.87	+0.63
	precision	0.80	0.59	-0.21	0.63	0.58	-0.05	0.74	0.55	-0.19	0.74	0.57	-0.17
Poi	Cost	102	102	-	96	96	-	120	120	-	104	104	-
	$recall_{defects}$	0.67	0.89	+0.22	0.64	0.90	+0.26	0.92	1.00	+0.08	0.58	0.92	+0.34
	precision	0.87	0.63	-0.16	0.73	0.63	-0.10	0.77	0.63	-0.14	0.81	0.64	-0.17
Prop	Cost	76	76	-	107	107	-	112	112	-	104	104	-
	$recall_{defects}$	0.67	0.92	+0.25	0.67	0.98	+0.31	0.91	0.99	+0.08	0.84	0.96	+0.12
	precision	0.67	0.62	-0.05	0.81	0.63	-0.19	0.77	0.61	-0.16	0.83	0.59	-0.24
Tomcat	Cost	214	214	-	241	241	-	64	64	-	54	54	-
	$recall_{defects}$	0.86	0.83	-0.03	0.56	0.83	+0.27	0.28	0.28	-	0.11	0.22	+0.11
	precision	0.22	0.08	-0.14	0.22	0.09	-0.13	0.58	0.04	-0.54	0.59	0.04	-0.55
Xalan	Cost	285	285	-	158	158	-	429	429	-	428	428	-
	$recall_{defects}$	0.38	0.70	0.32	0.37	0.78	+0.41	1.00	1.00	-	0.99	1.00	+0.01
	precision	1	0.99	-0.01	0.99	0.99	-	0.99	0.99	-	0.99	0.99	-

SO-C: Single-Objective Cross-project; **SO-W**: Single-Objective Within-project; **MO**: Multi-Objective

better prioritize classes with more defects than the single-objective models. Indeed, for the logistic model, at same level of cost MODEP classifies as defect-prone classes having more defects than the classes classified as defect-prone by the within-project single objective logistic regression. For 4 out of 10 projects (*Ant*, *Camel*, *Ivy*, and *Tomcat*), the $recall_{defects}$ is (about) the same, while for the other projects the difference in favor of MODEP ranges between +1% and +48% in terms of $recall_{defects}$ for the same amount of source code to analyze (KLOC), mirroring an increase of the number of defects contained in the classes identified as defect-prone ranging between +11 and +121 defects. The only exception to the rule is represented by *Ant* where the within-project single-objective predictor identifies a higher number of defects as compared to MODEP. Once again, MODEP provides an improvement in number of defects but also a general (often slight) decrement of precision. For what concerns the decision trees, we can observe that the results achieved are even better. MODEP predicts classes having more defects than those predicted by the single-objective within project prediction for 8 out of 10 projects. In this case, the difference in terms $recall_{defects}$ ranges between +1% and +63% with a corresponding increase of number of defects ranging between +3 and +401. Also for the decision tree predictor, there is a slight decrement of the precision for all the projets (lower than 6% in 50% of cases).

Also in this case, the statistical analysis confirmed our initial findings. Considering the logistic regression, the *precision* is significantly lower (p-value = 0.02) while, at the same cost, the $recall_{defects}$ significantly increases (p-value = 0.02). Similarly, for the decision tree we also obtained a significant decrement in terms of *precision* using MODEP (p-value = 0.03) but at the same inspection cost value we also achieved a statistically significant increase of $recall_{defects}$ (p-value < 0.01). Thus, we can reject both the null hypotheses, H_{0_R} in favor of MODEP and H_{0_P} in favor of the single-objective predictors.

4.5.2 RQ_2: *How does MODEP perform compared to the local prediction approach?*

Table 4.4: MODEP vs. local predictors when optimizing inspection cost and number of defect-prone classes predicted.

System	Metric	Local	MO-Logistic		MO-DTree	
Ant	Cost	132	132	-	132	-
	$Recall_{Classes}$	0.62	0.74	+0.12	0.66	+0.04
	Precision	0.32	0.28	-0.04	0.17	-0.15
Camel	Cost	68	68	-	68	-
	$Recall_{Classes}$	0.34	0.33	-0.01	0.34	-
	Precision	0.26	0.28	+0.02	0.17	-0.15
Ivy	Cost	59	59	-	59	-
	$Recall_{Classes}$	0.68	0.78	+0.10	0.78	+0.10
	Precision	0.25	0.20	-0.05	0.10	-0.15
jEdit	Cost	104	104	-	104	-
	$Recall_{Classes}$	0.56	0.97	+0.41	0.85	+0.44
	Precision	0.46	0.24	-0.22	0.22	-
Log4j	Cost	28	28	-	28	-
	$Recall_{Classes}$	0.52	0.94	+0.42	0.94	+0.42
	Precision	0.94	0.92	-0.02	0.92	-0.02
Lucene	Cost	73	73	-	73	-
	$Recall_{Classes}$	0.42	0.95	+0.43	0.95	+0.43
	Precision	0.80	0.58	-0.22	0.59	-0.21
Poi	Cost	85	85	-	85	-
	$Recall_{Classes}$	0.62	0.66	+0.64	0.93	+0.43
	Precision	0.88	0.62	-0.26	0.59	-0.21
Prop	Cost	91	91	-	91	-
	$Recall_{Classes}$	0.66	0.96	+0.30	0.94	+0.28
	Precision	0.85	0.63	-0.21	0.62	-0.23
Tomcat	Cost	201	201	-	201	-
	$Recall_{Classes}$	0.68	0.81	+0.13	0.73	+0.05
	Precision	0.22	0.08	-0.14	0.08	-0.14
Xalan	Cost	201	201	-	201	-
	$Recall_{Classes}$	0.68	0.81	+0.13	0.73	+0.05
	Precision	0.22	0.08	-0.14	0.08	-0.14

Local: Local Prediction; **MO**: Multi-Objective

Table 4.5: MODEP vs. local predictors when optimizing inspection cost and number of defects predicted.

System	**Metric**	**Local**	**MO-Logistic**		**MO-DTree**	
Ant	Cost	132	132	-	132	-
	$Recall_{Defects}$	0.66	0.63	-0.03	0.58	-0.08
	Precision	0.32	0.18	-0.13	0.64	+0.32
Camel	Cost	68	68	-	68	-
	$Recall_{Defects}$	0.46	0.49	+0.03	0.47	+0.01
	Precision	0.26	0.29	+0.03	0.29	+0.03
Ivy	Cost	59	59	-	59	-
	$Recall_{Defects}$	0.48	0.66	+0.18	0.66	+0.18
	Precision	0.25	0.09	-0.16	0.14	-0.11
jEdit	Cost	104	104	-	104	-
	$Recall_{Defects}$	0.56	0.97	+0.39	0.97	+0.39
	Precision	0.46	0.24	-0.22	0.24	-0.22
Log4j	Cost	28	28	-	28	-
	$Recall_{Defects}$	0.56	0.90	+0.34	0.91	+0.35
	Precision	0.94	0.92	-0.02	0.92	-0.02
Lucene	Cost	73	73	-	73	-
	$Recall_{Defects}$	0.57	0.99	+0.42	0.87	+0.44
	Precision	0.80	0.59	-0.21	0.56	-0.24
Poi	Cost	85	85	-	85	-
	$Recall_{Defects}$	0.65	0.85	+0.20	0.87	+0.22
	Precision	0.88	0.63	-0.25	0.59	-0.21
Prop	Cost	91	91	-	91	-
	$Recall_{Defects}$	0.70	0.89	+0.19	0.87	+0.17
	Precision	0.85	0.63	-0.22	0.69	-0.16
Tomcat	Cost	201	201	-	201	-
	$Recall_{Defects}$	0.71	0.75	+0.04	0.78	+0.07
	Precision	0.22	0.08	-0.14	0.13	-0.09
Xalan	Cost	201	201	-	201	-
	$Recall_{Defects}$	0.73	0.75	+0.03	0.73	+0.05
	Precision	0.22	0.08	-0.14	0.08	-0.14

Local: Local Prediction; **MO**: Multi-Objective

In this section we compare the performance of MODEP with an alternative method for cross-project predictor, i.e., the “local” predictor based on clustering proposed by Menzies *et al.* [86]. Table 4.4 shows $recall_{class}$ and *precision*—for both approaches—at the same level of inspection cost. Results indicate that, at the same level of cost, MODEP is able to identify a larger number of defect-prone classes (higher $recall_{class}$ values). Specifically, the difference in terms of $recall_{class}$ ranges between +13% and +64%. There is only one exception, represented by *Camel*, for which the $recall_{class}$ value is slightly worse (-1%). We can also note that in the majority of cases MODEP achieves a lower precision, ranging between 2% and 22% except for *Jedit* where it increases of 1%.

These findings are supported by our statistical analysis. Specifically, the Wilcoxon test indicates that the differences are statistically significant (p-value < 0.01) for both $recall_{class}$

and *precision*. Such results are not surprising since the local prediction model where designed to increase the prediction accuracy over the traditional (global) model in the context of cross-project prediction.

Table 4.4 shows $recall_{defect}$ and *precision*—for both MODEP and the local prediction approach—at the same level of cost. We can observe that MODEP is able to identify a larger number of defects (higher $recall_{defect}$ values), mirroring a better ability to identify classes having a higher density of defects. The difference, in terms of number of defects, ranges between +3% and +44%. There is only one exception, represented by *Ant*, for which the $recall_{defect}$ value is slightly lower (-3%). At the same time, the results reported in Table 4.4 also show that, in the majority of cases, MODEP achieves a lower precision, raging between -3% and -25%, with the only exception of *Camel* where it increases by 3%. Also in this case, the Wilcoxon test highlight that the differences in terms of $recall_{defect}$ and *precision* are statistically significant (p-value < 0.01).

4.5.3 Benefits of MODEP as Compared to Single-Objective Defect Predictors

Besides the quantitative advantages highlighted in $\mathbf{RQ}_1$, including the performance achieved for cross-project prediction, MODEP has also the advantage of providing the software engineer with the possibility to make choices to balance between prediction effectiveness and inspection cost.

A single-objective predictor (either logistic regression or decision trees) provides a single model that classifies a given set of classes as defect-prone or not. This means that, given the results of the prediction, the software engineer should inspect all classes classified as defect-prone. Depending on whether the model favours a high recall or a high precision, the software engineer could be asked to inspect a too high number of classes (hence, an excessive inspection cost), or the model may fail to identify some defect-prone classes. Truly, a mono-objective model could still rank classes based on their defect-proneness likelihood (e.g., estimated with a logistic regression model). However, even in this case a multi-objective model has the advantage of allowing the software engineer to choose the most suitable trade-off between cost and effectiveness.

Let us now consider some examples from the projects we studied in our study. In some of them—such as *Ivy* or *Tomcat*—a cross-project single-objective logistic regression reaches a high recall and a low precision, whereas for others—such as *Log4j* or *Xalan*—it yields a relatively low recall and a high precision. MODEP, instead, allows the software engineer to understand, based on the amount of code she can analyze, the level of recall (i.e., the percentage of defect-prone classes identified) that can be achieved, and therefore to select the most suitable predictors. For example, for *Xalan*, achieving a recall of 80% instead of 38% means analyzing about 132 KLOC instead of 13 KLOC (see Figure 4.5-e). In this case, the higher additional cost can be explained because most of the *Xalan* classes are defect-prone; therefore achieving a good recall means analyzing most of the system, if not the entire system. Let us suppose, instead, to have only a limited time available to perform code inspection. For

Table 4.6: The first 10 Pareto optimal models obtained by MODEP (using logistic regression) on *Log4j*.

Training Set		**Test Set**		**Logistic Coefficients**							
Cost	$Recall_{classes}$	*Cost*	$Recall_{classes}$	Scalar	*WMC*	*DIT*	*NOC*	*CBO*	*RFC*	*LCOM*	*LOC*
0	0%	0	0%	-0.91	0.26	-0.03	-0.55	-0.08	-0.06	-0.11	-0.65
0	0%	0	0%	-0.84	0.04	-0.02	-0.14	-0.01	-0.02	0.05	-0.40
2,330	1%	0	0%	-0.52	0.05	-0.01	-0.29	-0.11	-0.09	-0.01	-0.45
8,663	3%	46	6%	-0.47	-0.30	-0.02	-0.10	0.03	0.04	-0.03	-0.39
10,033	3%	133	6%	-0.42	0.10	-0.02	-0.25	-0.03	-0.04	-0.06	-0.73
18,089	6%	401	18%	-0.41	0.13	-0.01	-0.34	-0.03	-0.05	-0.08	-0.50
25,873	8%	512	20%	-0.36	-0.03	-0.02	0.00	0.04	-0.02	0.13	-0.56
33,282	8%	608	22%	-0.32	-0.06	-0.02	-0.06	0.04	-0.03	0.11	-0.49
43,161	10%	837	26%	-0.32	-0.03	-0.02	0.01	0.04	-0.02	0.12	-0.55
57,441	13%	1384	31%	-0.32	-0.05	-0.01	0.00	0.03	-0.01	0.02	-0.54

Ivy, as indicated by the multi-objective decision tree predictor, we could choose to decrease the inspection cost from 32 KLOC to 15 KLOC if we accept a recall of 50% instead of 83% (see Figure 4.4-c).

A similar analysis can be performed when considering the number of defects instead of the number of defect-prone classes as measure of the effectiveness of the prediction models. Specifically, MODEP allows the software engineer to analyze the trade-off between cost and percentage of defects contained in the classes identified as defect-prone, and then to select the predictors that best fit the practical constraints (e.g., limited time for analyzing the predicted classes). For example, let us assume to have enough time/resources to inspect the *Xalan* source code. In this case, by selecting a predictor that favors recall over precision, we can achieve a recall (percentage of defects) of 80% instead of 40% by analyzing about 200 KLOC instead of 50 KLOC (see Figure 4.7-e). Let us suppose, instead, to have only a limited time available to perform code inspection. For *Ivy*, as indicated by the multi-objective decision tree predictor, we could choose to decrease the inspection cost from 80 KLOC to 40 KLOC if we accept to identify defect-prone classes containing 50% of the total amount of defects instead of 83% (see Figure 4.6-c).

A further interesting property of MODEP is that software engineers can understand the relationship between the predictors and the desired outcome, i.e., inspection cost and prediction effectiveness, and how small changes of the coefficients affect the prediction. For example, let us consider the first 10 solutions belonging to the Pareto front obtained when using the multi-objective logistic regression to predict the defect-proneness of the classes for *Log4j* (the training set is composed by all the other projects). Table 4.6 reports the decision coefficients of the 10 solutions belonging to the Pareto front and the corresponding performance on both training and test sets. The data shows that the DIT and RFC predictors slightly affect the outcome since their coefficients (weights) are smaller than those of the remaining predictors. Furthermore, their small variation does not seem to affect the results (the variation is lower than ± 0.01). Conversely, the scalar value is strongly related to the outcome: the higher the scalar value, the higher the number of the classes predicted as defect-prone, and then the higher the $recall_{class}$ values. Also, the LOC predictor has the largest

absolute value, therefore it is the most influential predictor for the logistic models. A similar analysis can be performed on the decision rules that can be obtained from the multi-objective decision tree. However, since they produce more than 300 decision rules (coefficients), they are not reported for the sake of space. Last, but not least, it is important to point out that such a kind of analysis cannot be performed using the single-objective approach since they provide only one prediction model. Thus, for the single-objective approach it is impossible to determine how performances change varying the predictor variable coefficients and it does not help the software engineer to understand the relationship between predictors and outcome (prediction).

In summary, we can conclude stating that MODEP allows the software engineer to choose predictor that better suit the need for maximizing the amount of defect-prone classes to inspect (or the amount of classes with the highest number of defects), given the time/resources available. Also, by looking at the matrix of coefficients (e.g., Table 4.6) it is possible to understand what predictor variables lead towards a higher cost and/or a higher recall.

4.6 Threats to Validity

This section discusses the threats that could affect the validity of MODEP evaluation and of the reported study.

Threats to *construct validity* concern the relation between theory and experimentation. Some of the measures we used to assess the models (precision and recall) are widely adopted ones. We computed recall in two different ways, i.e., (i) as percentage of defect-prone classes identified as such by the approach, and (ii) as percentage of defects the approach is able to highlight. In addition, we use the LOC to be analyzed as a proxy indicator of the analysis/testing cost, as also done by Brahman *et al.* [88]. We are aware that such a measure is not necessarily representative of the testing cost especially when black-box testing techniques or object-oriented (e.g., state-based) testing techniques are used. Also, another threat to construct validity can be related to the used metrics and defect data sets. Although we have performed our study on widely used data sets from the PROMISE repository, we cannot exclude that they can be subject to imprecision and incompleteness.

Threats to *internal validity* concern factors that could influence our results. We mitigated the influence of the GA randomness when building the model by repeating the process 30 times and reporting the median values achieved. Also, it might be possible that the performances of the proposed approach and of the approaches being compared depend on the particular choice of the machine learning technique. We evaluated the proposed approach using two machine learning techniques—logistic regression and decision trees—that have been extensively used in previous research on defect prediction (e.g., Basili *et al.* [235] and Gyimothy *et al.* [243] for the logistic, Zimmermann *et al.* [85] for the decision tree). We cannot exclude that specific variants of such techniques would produce different results, although the aim of this Chapter is to show the advantages of the proposed multi-objective approach, rather than comparing different machine learning techniques.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. In addition to showing values of cost, precision and recall, we have also statistically compared the various model using the Wilcoxon, non-parametric test, indicating whether differences in terms of cost and precision are statistically significant.

Threats to *external validity* concern the generalization of our findings. Although we considered data from 10 projects, the study deserves to be replicated on further projects. Also, it is worthwhile to use the same approach with different kinds of predictor metrics, e.g., process metrics or other product metrics.

4.7 Conclusion and Future Work

This chapter proposed a novel formulation of the defect prediction problem. Specifically, it proposed to shift from the single-objective defect prediction model—which recommends a set or a ranked list of likely defect-prone artifacts and tries to achieve an implicit compromise between cost and effectiveness—towards multi-objective defect prediction models. The proposed approach, coined as MODEP (**M**ulti-**O**bjective **DE**fect **P**redictor), produces a Pareto front of predictors (in our work a logistic regression or a decision tree, but the approach can be applied to other machine learning techniques) that allows to achieve different trade-offs between the cost of code inspection—measured in terms of KLOC of the source code artifacts (class)—and the amount of defect-prone classes or number of defects that the model can predict (i.e., recall). In this way, for a given budget (i.e., LOC that can be reviewed or tested with the available time/resources) the software engineer can choose a predictor that (a) maximizes the number of defect-prone classes tested (which might be useful if one wants to ensure that an adequate proportion of defect-prone classes has been tested), or (b) maximizes the number of defects that can be discovered by the analysis/testing.

MODEP has been applied on 10 datasets from the PROMISE repository. The results indicated that:

1. While cross-project prediction is worse than within-project prediction in terms of precision and recall (as also found by Zimmermann *et al.* [85]), MODEP allows to achieve a better cost-effectiveness than single-objective predictors trained with both a within- or cross-project strategy;

2. MODEP outperforms a state-of-the-art approach for cross-project defect prediction [86], based on local prediction among classes having similar characteristics. Specifically, MODEP achieves, at the same level of cost, a significantly higher recall (based on both the number of defect-prone classes and the number of defects).

3. Finally, instead of producing a single predictor MODEP, allows the software engineer to choose the configuration that better fits her needs, in terms of recall and of amount of code she can inspect. In other words, the multi-objective model is able to tell the software engineer how much code one needs to analyze to achieve a given level of recall. Also, the software engineer can easily inspect the different models aiming

at understanding what predictor variables lead towards a higher cost and/or a higher recall.

In summary, MODEP seems to be particularly suited for cross-project defect prediction, although the advantages of such a multi-objective approach can also be exploited in within-project predictions.

Future work aims at considering different kinds of cost-effectiveness models. As said, we considered LOC is a proxy for code inspection cost, but certainly is not a perfect indicator of the cost of analysis and testing. Alternative cost models, better reflecting the cost of some testing strategies (e.g., code cyclomatic complexity for white box testing, or input characteristics for black box testing) must be considered. Last, but not least, we plan to investigate whether the proposed approach could be used in combination with—rather than as an alternative to—the local prediction approach [86].

Part II

Software Testing

Chapter 5

Software Testing: Background and Related Work

Contents

5.1 Introduction 118

5.2 Test data generation 119

5.2.1 Structural testing and basic concepts 120

5.2.2 Coverage criteria and flag problem 122

5.2.3 Solution Encoding for Test Data Generation 124

5.2.4 Fitness functions for test data generation 125

5.3 Test suite optimization 128

5.3.1 Problems definition 129

5.3.2 Test Suite Optimization with Traditional Approaches 131

5.3.3 Search-based Test Suite Optimization 132

5.4 Diversity and population drift 133

5.5 Improving evolutionary testing by diversity injection 136

5.1 Introduction

After changes are made to the software, it is necessary to test the new components added during the maintenance activities or re-testing the unchanged parts. Thus, software testing plays an important role in both software development and maintenance because it allows to gain confidence that the software behaves as intended. *Regression testing* is performed by re-running past test cases in order to verify whether changes have not endangered the system behavior and the system integrity has not been compromised. During this validation and verification process, some past test cases have to be selected, others have to be updated (*test suite optimization*). Finally, new test cases should be written in order to test new software components developed during the development process or during software maintenance activities (*test data generation*). In general, writing and updating test cases is a human-based process: testers have to read the specifications, decide what to test and which strategy to use when writing test case. Thus, this process is extremely costly, difficult and error-prone.

In order to reduce the high cost of manual test data generation and at the same time to increase the reliability of the testing processes researchers have tried to automate it. The simplest way to make such an automatism consists of testing/running a software program by considering all possible combinations of data inputs and preconditions, but it is impracticable/unfeasible. For these reasons many researchers reformulate the problem of test data generation as an optimization problem in order to guide the generation of input data according to specific *testing criteria*. The choice of the testing criterion to consider depends on the purpose and the type of testing being automated. For example, the tester would generate test cases in order to verify non-functional properties, such as the execution times (*non-functional criteria*). For safety tests, she would be interested in generating test cases for simulating error conditions derived from pre- and post-conditions (*safety criteria*). For automating structural testing, test cases are generated in order to maximize the number of executed elements of the program structure, thus, using some *structural testing* criteria. All works on these search-based topics are generally referred to as *Search Based Software Testing* (SBST) and nowadays they cover 54% of the overall literature of SBSE [47, 48, 49, 50].

Automatic techniques have been also used for regression testing, whose goal is to verify that software that has been modified still continues to match the specification and new changes have not introduced errors into unchanged parts, endangering their behaviors [27]. To this aim, an automated tool can be used to re-testing the whole software system by executing all the existing test cases in the test suite. Then, an automated comparison of the testing results obtained before and after changes (before and after a maintenance activities) can help to automatically identify defects inadvertently introduced by changes. However, since during the software life cycle the test suite tends to grow, this strategy might be too expensive and unfeasible, especially for large systems. For example, Do *et al.* [266] reported that a regression test suite containing over 30,000 functional test cases, requires over 1,000 machine hours to execute. This estimation considers only the running time of the regression testing and does not consider the time required by testers to oversee this regression testing process, set up test runs, monitor testing results, and maintain testing resources such as test

cases, oracles, and automation utilities.

To reduce the effort of regression testing several strategies have been proposed in literature by optimizing existing test suites in different ways [27]. A first way to reduce the cost of regression testing, named as *test suite minimization* or *test suite reduction*, consists of reducing the size of the test suite by removing test cases that are redundant with respect to some testing criteria, such as code coverage [34, 267, 35, 268, 97, 269, 104, 65]. A second strategy is the *test case selection* which is aimed at selecting a subset of the test cases that can be used to test the changed parts of the software under test [99, 98, 270, 68, 11]. Finally, *test case prioritization* is another strategy to optimize an existing test suite by ordering the test cases with the purpose of first executing those revealing faults earlier [96, 101, 102, 271, 68]. All these techniques are widely referred as to *test suite optimization* problems.

This second part of the thesis focuses on software testing with particular attention to test data generation and test suite optimization. Evolutionary algorithms, and GAs in particular, have been widely and extensively exploited in literature for trying to automate both these activities. Along with their strengths, testing techniques based on evolutionary algorithm have a set of challenges and open issues. We highlight that one of these issues is the genetic drift, or loss of diversity. Sections 5.2 provides a brief summary of the main related work and background notions about test data generation. Section 5.3 describes background and related work on regression testing. Section 5.4 highlights limitations and open issues of evolutionary testing techniques.

5.2 Test data generation

The earliest work in SBST was proposed by Miller and Spooner [39] in the context of structural testing for generating test data consisting of floating-point inputs. Their research direction inspired several later research works that proposed various techniques for automated test data generation, including symbolic execution [272, 273], random algorithms [274], constraint solving [275], the chaining method [276, 277], and evolutionary testing [47, 142, 28, 21]. All techniques for structural testing fall in two main categories: *static* and *dynamic* structural test data generation. Static techniques are based on the analysis of the internal structure of the program, without executing the program itself. For example, *symbolic execution* [272, 273, 278] consists of assigning symbolic values to variables in order to derive through mathematical/abstract methods the conditions necessary for traversing specific paths in the program [278, 279]. Although promising results have been obtained, these techniques have many problems [45]: scalability, non-linear predicates, non-primitive data types, loops, arrays, etc.

The opposite of static/symbolic testing consists of analyzing the behavior of the program during its execution with a given input. With *dynamic structural test* data generation the program under test is executed with some input and the results are analysed via program instrumentation. Duran and Ntafos [280] investigate the usage of random testing as dynamic approach. They simply executes the program with random inputs and then observes the

executed/covered elements in the program structure. Despite it simplicity, this techniques does not provide good performance where the number of inputs covering a particular structural element are very few in number compared to the size of the input domain. Thus, it is generally used as a baseline for evaluating the performance of other search algorithms. Chen *et al.* [281] proposed an adaptive version of this technique (ART) in order to improve the performances of random testing. However, a recent study by Arcuri and Briand [282] revealed that ART is inefficient even on trivial problems, preventing its practical use in most situations.

A way to make faster and more reliably an automated test data generation is to use some guidance measure to a search algorithm [67]. According to Ferguson and Korel [277] this guidance can be provided in the form of a problem-specific fitness function, which can be utilized by optimization algorithms, such as Hill Climbing, Simulated Annealing or Evolutionary algorithms [67]. As result, a large body of research works has been reported where different meta-heuristic techniques have been applied to address the problem of test data generation [45]. The application of Evolutionary Algorithms to test data generation, often referred to as Evolutionary Testing, has been intensively investigated for the purpose of structural testing [47, 142, 28, 21]. Among all evolutionary algorithms, in this thesis we focused on the application of GAs since they are particularly popular flavor of evolutionary algorithm quite often and successfully adopted by the testing community [47, 142, 28, 21].

As explained in Section 2.3, there are two key factors that have to be defined in order to apply GAs to test data generation problem: (i) solution encoding or representation of candidate solutions as chromosomes; (ii) defining a fitness function to guide the search toward promising areas of the search space. The next subsections provide background notions about structural analysis and solution encoding for software testing. It is also provides a brief discussion about some fitness functions that have been used by different authors in SBST.

5.2.1 Structural testing and basic concepts

A program P can be considered as a function $P : I \leftarrow O$, where I represents the set of all possible data inputs while O is the set of all possible outputs. In particular, I is the set of all values that are assigned to the input variables of P, where an input variable of P is a variable that either appears as input parameter of P or in a its input statement (e.g., the variable `x` in Pascal statement `read(x)`). The execution of a program P with a specific data x (this process is usually denoted by $P(x)$), implies a specific execution sequence of structural elements in P (either branches and statements). Clearly, different data input might derive different output data and different execution sequences of elements in P. Therefore, the intuitive goal of SBST approaches is to generate data input (test cases) which allows to execute all the elements in P, thus, it is necessary to analyse the structure of P and its tested elements.

To make such an analysis, structural testing techniques use the *control flow graph* (CFG) which is a graphical representation of a program. A CFG for a program P is a directed graph $G = (N, E, s, e)$ where:

Nodes	Instructions
s	`int tri_type(int a, int b, int c)`
	`{`
1	`if (a > b)`
2-4	`{ int t = a; a = b; b = t; }`
5	`if (a > c)`
6-8	`{ int t = a; a = c; c = t; }`
9	`if (b > c)`
10-12	`{ int t = b; b = c; c = t; }`
13	`if (a + b <= c)`
	`{`
14	`type = NOT_A_TRIANGLE;`
	`}`
	`else`
	`{`
15	`type = SCALENE;`
16	`if (a == b && b == c)`
	`{`
17	`type = EQUILATERAL;`
	`}`
18	`else if (a == b || b == c)`
	`{`
19	`type = ISOSCELES;`
	`}`
	`}`
e	`return type;`
	`}`

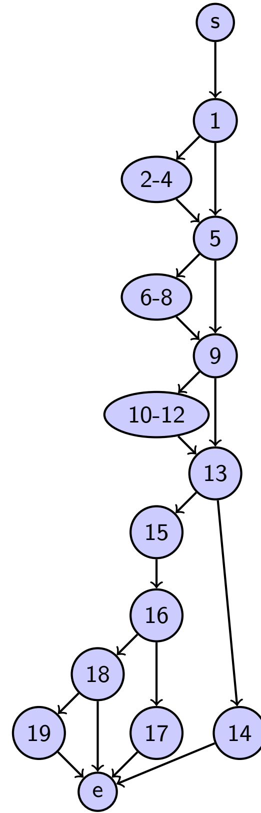


Figure 5.1: Example of triangle classification program

- N is a set of nodes in the graph. Each node $n \in N$ is a statement in the program or a basic block (a set of uninterrupted consecutive sequence of statements with no jumps).
- E is a set of edges in the graph. Each edge, $e = (n_i, n_j) \in E$, represents a transfer of control from node n_i to node n_j.
- s denotes the unique entry node to the graph.
- e is the unique exit node to the graph.

Nodes corresponding to decision statements (for an example an `if` or a `while` statement) are referred to as *branching nodes*. Outgoing edges from these nodes are referred to as *branches*. The condition determining whether a branch is traversed is referred to as the *branch predicate*. In order to traverse a specific edge it is necessary to hold the condition (branch predicate) of the corresponding outgoing edge.

An example of a control flow graph for the triangle classification program can be seen in Figure 5.1. This program takes as input three positive integer values (that represent the

lengths for the sides of a triangle) and decides if the triangle is isosceles, scalene, equilateral, or invalid. In the triangle example, branching nodes are nodes 1, 5, 9, 13, 16 and 18. The edge (16,17) is traversed if and only if the input values (`a`, `b`, `c`) satisfy the branch conditions of node 16 (`a==b` and `b==c`).

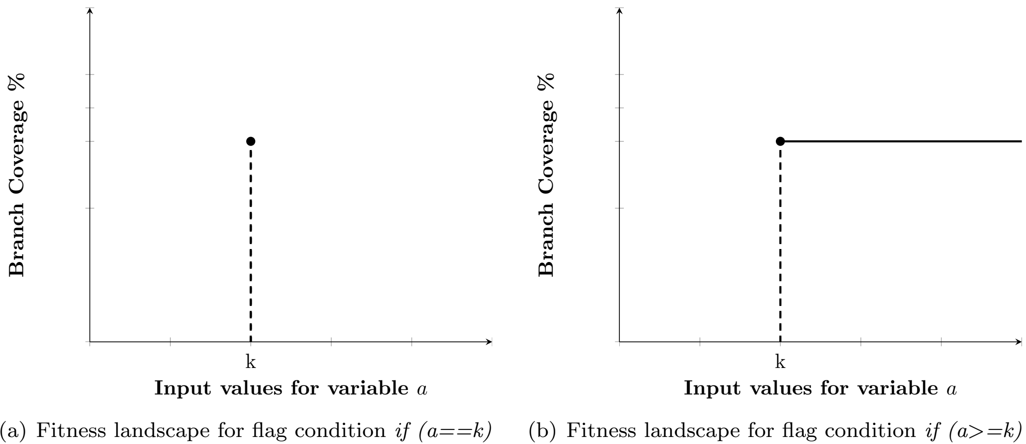
A path p through a CFG is a sequence of nodes $p = \langle n_1, n_2, \ldots, n_m \rangle$, such that each pair of consecutive nodes (n_i, n_{i+1}) are connected through a directed edge from n_i to n_{i+1}, i.e, $\forall i$, $1 \leq i < n$, $(n_i, n_{i+1}) \in E$. Whenever the execution of $P(x)$ traverses all the edges in a path p, we say that x traverses/covers p. A path is said to be *feasible* if there exists an input for which the path is traversed, otherwise the path is said to be *infeasible*. If a path begins with the entry node and ends with the exit node is called as *complete path*. Otherwise it is called as *incomplete path* or a path segment. For the triangle program in Figure 5.1 the sequence $p = \langle s, 1, 5, 9, 13, 14, s \rangle$ is a feasible and complete path within the corresponding CFG (this path can be traversed using the following input data: `a=0`, `b=0` and `c=0`).

A *definition node* is a node in the CFG that modifies the value of a program variable v, e.g, nodes containing an assignment statement or an input statement. In the example of Figure 5.1, nodes 14, 15, 17, 19 are definition nodes for the variable `type`. An *use node* in a CFG is a node that uses a variable program v, e.g, nodes containing an assignment statement, an output statement, or a branch predicate expression. For the example in Figure 5.1, node 1 is an use node for the variable `a`. A *definition-clear path* with respect to variable program v is a path that does not modify v (it contains nodes/statements not modifying v). In the triangle example, the path $p = \langle 13, 15, 16, 17 \rangle$ is a definition-clear path for the variables `a`, `b` and `c`.

5.2.2 Coverage criteria and flag problem

The CFG representation is used to keep track of the program elements that are executed (traversed) during the execution of the program itself. Thus, CFG and dynamic analysis can be used to rigorously describes which parts of program are tested. This has led toward a definition of many *test coverage* metrics according to the type of code elements that should be covered/tested. Below is reported a list of the most used coverage criteria [283, 284]:

- *Statement coverage.* With this criterion, the goal of structural testing is to execute all statements in the CFG.
- *Branch coverage*, which aims at executing all branches in the CFG. This means that the condition of an `if` statement must be evaluated to both `true` and `false` branches.
- *Condition coverage.* Using this criterion, all clauses within each condition of the CFG must be executed to both `true` and `false` branches.
- *Multiple-condition coverage.* All combinations of clauses within each condition of the CFG must be executed using truth values.
- *Path coverage.* With this criterion, the goal of structural testing is to traverse all paths in the CFG.



(a) Fitness landscape for flag condition *if (a==k)* (b) Fitness landscape for flag condition *if (a>=k)*

Figure 5.2: Example of flag landscape. The y-axis measures the branch coverage while the x-axis represents the input space.

Statement coverage is the weakest criterion because it does not reveal faults that can be exposed by specific paths in the CFG (or specific sequences of statements). The other criteria are stronger then statement coverage, however, some of them can be unfeasible for large/complex program (e.g., path coverage and condition-coverage criteria). For these reasons the branch coverage criterion is the most used criterion in structural testing [45].

Coverage criteria are the most intuitive fitness functions that could be considered when formulating the problem of automatic test data generation as optimization problem: *finding the set of data input to a program P in order to maximize a coverage criterion.* However, despite its simplicity this formulation does not provide any guidance to the search algorithm for structures, which are generally covered by chance. A typical example to explain this problem is represented by branch predicates that are only true when an input variable has to be a specific value from a large domain [45]. For example, suppose we want to test the true branch of the condition statement `if (a==k)`, where `a` is an input variable while `k` is a numeric constant value. In this case within the space of all possible input values for `a`, there is only one value that allows to traverse the true branch of the condition (*flag problem*). If we use the branch coverage criteria as fitness function there is no guidance from lower fitness to higher fitness as shown in Figure 5.2-a. The flag problem is even presents when considering a less strong condition (e.g., `if (a>=k)`). As shown in Figure 5.2-b, in this case the corresponding fitness landscape is a step function with no guidance, thus, the true branch can be covered only by chance.

To overcome flag problems, previous works [279, 285, 39] have suggested to define the fitness function in terms of program predicates and using coverage criteria as stop-condition for a search process: *generate input data (test cases) until the maximum coverage is reached or the coverage is not improved further within a specific amount of time.*

5.2.3 Solution Encoding for Test Data Generation

As any SBSE approach to solve an engineering problem, the first step consists of encoding a candidate solution for test data generation in a form that is suitable for search algorithms, and GAs in particular. In our case, a candidate solution is a test case, which is represented as a set of input values (input vector) for input variables of the program P to be tested. Input variables $x_1, \ldots x_k$ are variables that either appear as input parameter of P or in a its input statement. More formally, an input vector I is a vector $I = (x_1, x_2, \ldots, x_k)$ of input variables to P. The domain of a generic input variable x_i, denoted as D_{xi} is the set if all possible values that x_i can assume (i.e, D_{xi} characterizes the feasible region for the variable x_i). Then, the domain of a program P is the cross product of the domains of all input variables $D_P = D_{x1} \times D_{x2} \times \cdots \times D_{xk}$. Thus, a candidate solution to a program P is a single instance of the program domain and can be viewed as a single point in a k-dimensional space. This representation of candidate solutions lends itself to be encoded as chromosome for genetic algorithms.

In a unit testing scenario, a test case t is not only an input vector I but it is a program that executes the program P with input I. Thus, other than containing the input data, a test case requires a set of statements (instructions) written in a target language (e.g., in Java) that allows one to encode optimal solutions for the addressed problem. Thus, a test data generator tool must be able to map an *input vector* with the corresponding statements that use such input variables in the test case. Typically, evolutionary algorithms have no understanding of programs, statements, objects, and so on. Therefore, a means of encoding must be defined which allows the representation of a test program as a basic type value structure with which an evolutionary algorithm can work [286].

According to [287, 142] a test case is a sequence of statements $t = \{s_1, s_2, \ldots, s_l\}$ of length l. A statement consists of the following essential components: (i) target object, (ii) methods and (iii) parameters/vaiables. These are the only information that should be encoded in the genotype of candidate solutions. For object-oriented program languages, each statement in a test case belongs to one of the following five different categories [287, 142]:

- *Primitive statements* represent numeric, boolean, string variables, as for example `int var = 1`. Other primitive statements are those defining arrays of any type (e.g., `Object[] var1 = new Object[10]`). For these primitive statements a test data generator tool have to define/store the values and the types of variables. An array definition also involves the definition of a set of values of the component type of the array, whose number is equal to the length of the array.

- *Constructor statements* generate new instances of any given class; e.g., `Object obj = new Object()`. For these statement a test data generator tool have to define/store the type and the values of the input parameters/values of the constructor.

- *Field statements* access public member variables of objects, e.g., `int var = l.size`. For these statements values and type have to be managed.

- *Method statements* invoke methods on objects or call static methods, e.g., `boolean flag = string.substring(3)`. For these statements values, type and returned values have to be managed.

- *Assignment statements* assign values to array indices or to public member variables of objects, e.g., `var[0] = new Object()` or `var.attribute = 2`.

For each of the above statement type the solution encoding schema have to store input data values (e.g., numeric values, string, etc.) and language based information (target object, methods name, parameters name, etc.). A detailed description of the mechanism that can be used to manage the encoding and decoding of target objects, methods and type of parameters can be found in [286], while further details about the chromosome representation of candidate solutions in object-oriented programming can be found in [28]. For these solution encoding schemata, the genetic operators (crossover and mutation operators) should be able to modify not only the input data values but also the language based information (e.g, by replacing a call to a method m_1 with a call to another method m_2 in a specific statement of the encoded test case). Further details on genetic operators for evolutionary test data generation can be also found in [28].

5.2.4 Fitness functions for test data generation

To automate software test data generation using evolutionary algorithm, the problem must first be transformed into an optimization task. Basically, there are three approaches that differ in the type of information used when building the objective function: *branch-distance-oriented*, *control-oriented*, or *combined* approaches. In general, the fitness function is defined as a function f to minimise, where $f(t) = 0$ if and only if the target branch is covered when a test case t is executed.

Branch-distance oriented approach

With branch distance approach the fitness function f is typically defined in terms of program predicates. Specifically, it measures how distant the executed control flow is away from traversing the desired (target) path or branch when using a given input data. For example, consider a branch condition `if (a==b)` that needs to be executed traversing the true branch; the condition is converted in a numerical objective function to be optimized using `abs(a-b)`. This objective function measures how the input data `a` and `b` are close to satisfy the true branch of the corresponding condition. The lower the branch distance value, the closer the true branch to being traversed, and the closer the input data to the required test data.

Tracey *et al.* [288, 289] proposed a set of objective functions for relational predicates inspired by the original objective functions defined by Korel [276]. Table 5.1 reports the branch distance for each type of possible conditions. In general, for each condition in the CFG the branch distance f takes as input a set of input values I, and it evaluates a numerical expressions on the basis of the actual values in I. This function f works fine for expressions

Table 5.1: Tracey's objective functions for relational predicates. k can be any arbitrary positive constant value.

Predicate	**Objective function** f
Boolean	if TRUE then 0 else K
$a == b$	if $abs(a-b)=0$ then 0 else $abs(a-b)+K$
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else K
$a < b$	if $a-b<0$ then 0 else $(a-b)+K$
$a \leq b$	if $a-b \leq 0$ then 0 else $(a-b)+K$
$a > b$	if $b-a<0$ then 0 else $(b-a)+K$
$a \geq b$	if $b-a \geq 0$ then 0 else $(b-a)+K$
not a	Negation is moved inwards and propagated over a
$A \wedge B$	$f(A)+f(B)$, where A and B are two clauses
$A \vee B$	$\min\{f(A), f(B)\}$, where A and B are two clauses

involving numbers (e.g., integer, float and double) and boolean values. For other types of expressions, such as an equality comparison of pointers/objects and strings, more complex branch distance functions have been defined in literature [67, 48, 290]. Finally, given a specific target path or branch to be traversed and a set of input values I, the final objective function is measured as the sum of all branch distance values encountered in the path traversed when running the program with input data I.

Given the branch distance as fitness function, a search algorithm is executed to find input data that satisfies a target element in the CFG at once, and the search process continues until all the elements are covered/traversed according to a specific coverage criterion (or the maximum coverage values cannot be further improved). Xanthakis *et al.* [285] used GAs to generate test data for paths in the CFG not covered by random search. In this work the selection of the path to be considered during the search process is performed by the tester. Jones *et al.* [26] removed the problem of selecting specific path by considering branch coverage and computing the fitness function as the branch distance of the required branch.

One of the main problems of the branch-distance oriented approach regards the presence of the flag problem for conditions involving a boolean value, then it is either false or true. For this kind of predicated there is no guidance (gradient) for search algorithms since the branch distance would assume only two different values. To alleviate the flag problem several authors have suggested to use *testability transformations* [47, 50] and more sophisticated fitness functions [49].

Control oriented approach

With control-oriented approach the fitness function is typically defined in terms of nodes that need to be executed in order to test the desired program element. Jones *et al.* [26] considered as objective function the difference between the number of actual and desired executed instructions. Pargas *et al.* [20] defined an objective function that measures the number of program elements executed as intended when running a program with a specific

Nodes	Instructions
s	`example(int i, int j)`
	`{`
1	`if (i >=10 && i<= 20)`
	`{`
2	`if (j >= 0 && j <= 10)`
	`{`
3	`// target statement`
	`}`
	`}`
	`}`

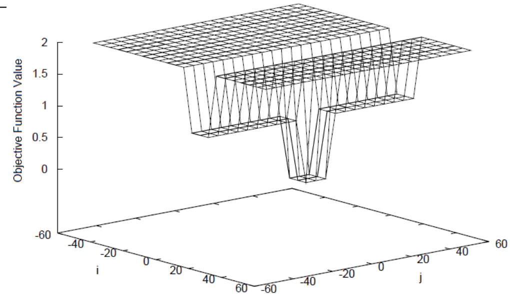


Figure 5.3: Objective function landscape by Pargas *et al.* [20] for a simple program.

input data. Thus, they used statement and branch coverage by analysing the structure of the CFG during program execution. As pointed out in [45], the measure used by Pargas *et al.* is equivalent to the number of *critical branches* successfully avoided by the input data. A critical branch for a given target node is the edge in the CFG which leads the execution path away from the target node. If control flow is driven down a critical branch, there is no prospect of the target being reached.

More formally, let I be the set of input values used to execute a program P; let *dependent* be the number of dependent nodes that should be executed for the current target element in the CFG; let *executed* be the number of dependent nodes successfully executed as intended when running P with input I. The objective function proposed by Pargas *et al.* [20], can be computed as follows:

$$\min f(I) = (dependent - executed) \tag{5.1}$$

A disadvantage of the objective function [26, 20] based on control-oriented analysis is that they do not give any guidance during the search process. Indeed, since the fitness function does not consider the distance of branch predicates, all input data executing the same dependent nodes in the CFG have the same fitness value. For example, let us to consider the simple program in Figure 5.3-a. Suppose we choose the node 3 as the target node. This node is control dependent on node 2 and node 1. The corresponding landscape obtained by applying the approach by Pargas *et al.* [20] has three plateaux (flat or step landscape), as shown in Figure 5.3-b. For input data satisfying the same branch predicates, no guidance is given.

Combined approach

To overcome the limitations of the previous approaches, Wegener *et al.* [21] proposed a combined approach that condenses branch distance and control-based approach in only one objective function. More formally, let I be the set of input values used to execute a program P; let *dependent* be the number of dependent nodes that should be executed for the current

Nodes	**Instructions**
s	`example(int i, int j)`
	`{`
1	`if (i >=10 && i<= 20)`
	`{`
2	`if (j >= 0 && j <= 10)`
	`{`
3	`// target statement`
	`}`
	`}`
	`}`

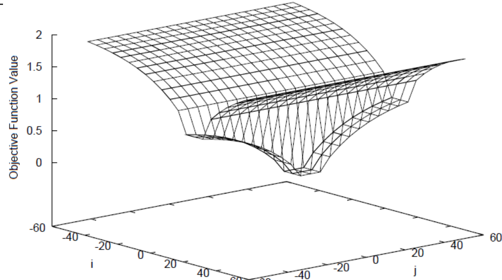


Figure 5.4: Objective function landscape by Wegener *et al.* [21] for a simple program.

target element in the CFG; let *executed* be the number of dependent nodes successfully executed as intended when running P with input I. The objective function proposed by Wegener *et al.* [21] has the following definition:

$$\min f(I) = (dependent - executed - 1) + m_branch_dist \tag{5.2}$$

The factor $(dependent - executed - 1)$ is named *approach level* and measure the number of the target's control dependent nodes that were not executed by the path for the input I. The approach level is the number of the target's control dependent nodes that were not executed by the path for a given input. When the execution of a test case diverges from the target branch at some approach level, the branch distance is computed. Specifically, the factor m_branch_dist is the branch distance value scaled in the range $[0, 1]$. This factor measures how close the input data I is to executing the required branch for descending the next approach level. For example, let us to consider the same program reported in Figure 5.3-a. If we consider the node 3 as target node, the resulting objective function landscape is shown in Figure 5.4. As we can see, adding the branch distance to the approach level prevents the formation of flat landscape. In this thesis we consider as objective function the combined approach by Wegener *et al.* [21], since it allows to overcome some limitations of the other objective functions previously described. The combination of approach level and branch distance is also the widely used objective function in literature for test data generation [21, 47, 142, 28, 21].

5.3 Test suite optimization

In general, solving test suite optimization problems involves (i) choosing a testing criterion to be satisfied and (ii) using an optimization technique to select/order the test cases on the basis of the chosen criterion. For example, widely used criteria are code coverage [96], program modification [100], execution cost [101], fault history information [68], etc. Hemmati

et al. [291] suggest to use test case diversity as test criterion to be optimized when selecting test cases. Recently, researchers have also suggested that a combination of multiple—often contrasting—criteria can be more useful than the individual criteria [97, 104, 68, 103, 105] for regression testing.

Once a set of test criteria has been chosen in a multi-criteria paradigm, an optimization algorithm has to be used to select/order the test cases. An exhaustive search of the subsets of test suite which are optimal according to multiple testing criteria should analyse 2^n possible subsets of the test suites, where n is the size of the original test suite. For example, a systems with 100 test cases has a huge search space represented by $2^{100} \approx 10^{30}$ possible sub-test suites to be analyzed. This number is much greater than the estimated age of the universe expressed in seconds[1].

Since the exhaustive search is unfeasible for practical purpose, approximation algorithms have been used to deal with regression testing in an acceptable time. All the approximation algorithms proposed in literature fall in two categories: deterministic or *classical* and *search-based* approaches. A complete survey of such approaches can be found in [27]. The following subsections provide a discussion of the most relevant related work and approaches. In particular, Section 5.3.1 provides a formal definition of regression testing problems. Section 5.3.2 provides an overview on traditional approaches to test suite optimization, while Section 5.3.3 focuses on search-based approaches.

5.3.1 Problems definition

Approaches aiming at reducing the effort of regression testing include *test suite minimization* (or reduction), *test case selection*, and *test case prioritization*. These approaches as generally referred to as *test suite optimization* approaches for regression testing.

Test Suite Minimization

The goal of the test suite minimization problem (TSM) consists of reducing the size of the test suite by deleting test cases that are redundant with respect to some coverage criteria, such as code coverage, branch coverage, data flow, dynamic program invariants or call stacks [34]. According to Rothermel *at al.* [268], the problem can be formalized as follows:

Definition 8. *Let $T = \{t_1, \ldots, t_n\}$ be a test suite; let $R = \{r_1, \ldots, r_m\}$ be a set of test requirements that must be satisfied with the aim at providing a desirable coverage of the program entities; let $\{T_1, T_2, \ldots, T_k\}$ be subsets of T one associated with each of the test requirement r_i such that any one of the test cases t_j belonging to T_i can be used to test the requirement r_i.*

***Problem**: find a representative set T' of test cases from T that satisfies all the test requirements $\{r_1, \ldots, r_m\}$.*

[1] According to recent estimation, the age of the universe is $(4.354 \pm 0.012) \times 10^{17}$ *seconds*.

Finding the minimal subset $T' \subseteq T$ which satisfies all the test requirements (e.g., code coverage criterion) is NP-complete, because the *minimal hitting set* problem can be reduced to it in polynomial time[2].

Test Case Prioritization

Test Case Prioritization (TCP) is aimed at ordering test cases for maximize some desired properties. It involves the execution of the test cases in a given order and terminating the testing process at some arbitrary point chosen by the decision maker [27]. The formal definition was provided in Rothermel et al. [293], which is the following:

Definition 9. *Let T be a test suite, and PT be a set of permutation of T; let $f : PT \longrightarrow \mathbb{R}$ be a function from PT to real number.*
Problem: *to find $T' \in PT$ such that $\forall\, T'', T'' \in PT, T'' \in PT, T'' \neq T', [f(T') \geq f(T'')]$.*

The function f is the mathematical description of the test criterion to be optimized, while PT is the set of all the possible permutation of T. The ideal ordering of test cases is the one that maximize the real fault detection rate but it can not be known to the tester in advance until the test case are executed in a given ordering. Hence, the ordering criteria generally depend on surrogates, which are in some way correlated with the fault detection rate. Code coverage is widely used as prioritization criterion [293, 101, 294], such as branch coverage, statement coverage, etc. The idea is that an ordering which executes first test cases with higher code coverage, also early covers the whole code increasing the probability of early detecting faults. Other prioritization criteria were also used instead the structural coverage, such as interaction [295], clustering-based [296], and requirement coverage [297].

Test Case Selection

Test case selection (TCS) focuses on selecting a subset from an initial test suite in order to test software modifications, i.e. to test whether unmodified parts of a program still continue to work correctly after changes involving other parts. The formal definition of the TCS problem, provided by Rothermel and Harrold [270], is the following:

Definition 10. *Let P be an application program and P' be a modified version of P and let T be the original test suite developed for testing P.*
Problem: *select a subset of test cases $T' \subseteq T$ with which to test P'.*

TCS problem essentially consists of two main steps: (i) identifying the affected but not modified parts, which typically involves a white-box static analysis of the program code [27]; and (ii) selecting test cases, which involves the identification of a subset of a test suite that is able to test the unmodified parts of a program (i.e., a minimized test suite that has the potential to detect defects introduced by changes).

[2] It is also the dual problem of the minimal set cover problem [292]

5.3.2 Test Suite Optimization with Traditional Approaches

With TSM strategy the size of the test suite is reduced by deleting test cases that are redundant with respect to some coverage criteria [268], such as code coverage, branch coverage, data flow, dynamic program invariants or call stacks [34]. Since the minimal subset of a test suite is NP-complete, several heuristics have been applied to deal with this problem [34, 35, 36]. Harrold *et al.* [34] used the traditional greedy algorithm for the minimal hitting set problem, with the worst case execution time of $O(\mid T \mid \cdot \max \mid T_i \mid)$, where $\mid T \mid$ represents the size of the original test suite, while $\max \mid T_i \mid$ denotes the cardinality of the largest group of test cases which is able to cover all the test requirements $\{r_1, \ldots, r_m\}$. This algorithm starts with an empty subset of the test suite and iteratively adds the test case that provides the higher coverage among the remaining test cases. The process will continue until the maximum coverage is not reached.

Since the set cover problem is the dual of the minimal hitting set [292], Chen and Lau [36] proposed a variant of the greedy algorithm that is known to be an effective heuristic for the Set Covering Problem (SCP). However, an empirical comparison between the two greedy approaches suggested that none of the two techniques is able to outperform the other. Offutt *et al.* also treated the test suite minimization problem as a SCP [35], proposing different variants of the greedy approach with different test cases ordering criteria instead of the fixed ordering of the original greedy approach [27]. Offutt *et al.* [35] used greedy approaches with different test case ordering criteria instead of the fixed ordering of the original one. An empirical comparison of greedy approaches suggested that none of them is able to outperform the others [35].

Further work based on greedy approaches considered other coverage criteria than the code-level structural coverage criteria used by Harrold *et al.* [34], Offutt *et al.* [35], and Chen *et al.* [36]. For example, Marré and Bertolino [298] formulated the TSM problem as the problem of finding a minimal spanning set over the *decision-to-decision* graph. McMaster and Memon [269] proposed a test suite minimization technique based on call-stack coverage. Black *et al.* [97] considered a bi-criteria approach that takes into account two testing criteria: (i) code coverage and (ii) past fault detection history. They combined the two objectives by applying a weighted-sum approach, and used Integer Linear Programming (ILP) optimization to find subsets, then reducing the multi-objective problem to a single-objective one. Finally, several studies have been focused on the effect of test suite minimization on the ability of a test suite to detect faults, since a smaller test suite might have a lower effectiveness. Rothermel *et al.* [299] have highlighted how the fault-detection ability of the reduced test suite was worsened, while Wong *et al.* [300] have shown that the fault detection ability of the reduced test suite was preserved.

In TCP the ordering criterion depends on surrogates which are in some way correlated with the fault detection rate, such as code coverage [101, 293, 294], interaction coverage [295], clustering-based coverage [296], and requirement coverage [297]. According to the chosen surrogate, the ordering of the test cases is computed using a greedy algorithm, since the ordering by which the test cases are selected by the algorithm also mirrors the ordering to

execute them. Greedy algorithms have also been used to solve a bi-criteria TCP problem by conflating two objectives (coverage and cost) in only one function (coverage per unit cost) to be maximized by applying the weighted-sum approach [101, 102]. Rothermel *et al.* [293] compared different prioritization techniques based on different coverage testing criteria (surrogates) to the random prioritization. An empirical study performed the Siemens benchmark demonstrates that the random prioritization is worse than the non-random orderings, where the performance are measured as the ability to early detect faults. This empirical study was extended by Elbaum *et al.* [301] considering further testing criteria with different granularity, such as statement coverage or function coverage. Do *et al.* [302] also provide a further empirical evidence on the usefulness of coverage-based prioritization against a random ordering of test cases using the java unit test framework (JUnit).

Test case selection (TCS) focuses on selecting a subset from an initial test suite to test software changes, i.e., to test whether unmodified parts of a program still continue to work correctly after changes involving other parts [270]. The identification of the modified parts of software can be performed using different techniques, including Integer Programming [99], symbolic execution [303], data flow analysis [98], dependence graph based techniques [267], and flow graph-based approaches [270]. The details of the different selecting approaches differ based on how a specific technique defines, seeks and identifies changes in the program under test [27]. Once the test cases covering the unmodified parts of programs are identified using a given technique, optimization algorithm—i.e., additional greedy—can be used to select a minimal set of such test cases according to some testing criteria—e.g., statement coverage—with the purpose of reducing the cost of regression testing.

As it has been previously pointed out by Yoo and Harman [68, 103], test suite minimization, test case selection and test case prioritization are strongly related to each others. For example, both test suite minimization and test case selection involve the selection of elements (test cases) from a test suite (starting set) that best satisfy the testing criteria (e.g., code coverage) [304]. Test case prioritization is also highly related to test case selection [68, 103, 100], since an optimal ordering can be applied to the test cases aiming at reducing the cost of regression testing. For instance, Srivastava and Thiagarajan [100] combine prioritization and test case selection. Specifically, they first detect the unmodified parts of software by comparing the binary code before and after changes. Hence, a greedy algorithm is used to order test cases but only according to the coverage of unmodified parts.

5.3.3 Search-based Test Suite Optimization

Test case selection, test suite minimization, and test case prioritization can be viewed as multi-objective problems, were the goal is to select a Pareto-efficient subset of the test suite, based on multiple test criteria [68, 103]. Recently, Sampath *et al.* [105] provide an analysis on the benefit of combining multiple criteria for regression testing, showing that the combined (hybrid) criterion often outperformed their constituent individual criteria.

Let $T = \{t_1, \ldots, t_n\}$ be a test suite and $F = \{f_1, \ldots, f_m\}$ a set of objective functions, i.e., the mathematical descriptions of test criteria to be satisfied during the selection of

test cases. The multi-objective test suite optimization problem can be defined as follows [68]: selecting a subset $T' \subset T$ such that T' is the Pareto-optimal set with respect to the objective functions in F. The optimality of solutions are measured by the concepts of Pareto optimality and Pareto dominance that are generally used in multi-objective optimization (see Section 2.3).

Yoo and Harman [68, 103] considered two and three contrasting test criteria: code coverage and execution time in the two-objective formulation; then, they added the fault history information as third criterion in the three-objective formulation. They also evaluated different optimization algorithms to find Pareto-optimal sub-sets of the test suite: additional greedy algorithms and a multi-objective genetic algorithms called NSGA-II [107]. The additional greedy algorithms were applied by using the traditional weighted sum approach to conflate all the objectives in only one function to be optimized: a cost cognizant version of the additional greedy algorithm was used for the two-objective formulation, while the weighted sum of code coverage per unit time and fault coverage per unit time was considered for the three-objective formulation. The empirical comparison between MOGAs and greedy algorithms did not reveal a clear winner among them, and in some cases the MOGAs were not able to outperform the greedy algorithms [68]. Furthermore, the combination between these two kinds of algorithms was not always useful to reach better results [103]. Also, greedy algorithms—which perform well for single-objective formulations—are not always Pareto-efficient in the multi-objective paradigm, motivating the use of meta-heuristic techniques [68, 103]. Similar considerations have also been provided by Li *et al.* [305], who investigated many meta-heuristics algorithms for the single-objective formulation, including hill climbing algorithms, GAs, additional greedy algorithms, and two-optimal greedy algorithms.

5.4 Diversity and population drift

There are two main factors in the search process based on GAs: *selective pressure* and *population diversity* [137]. The selective pressure is important for driving the GA toward an optimum, while population diversity is crucial for ensuring that the solution space is adequately explored. A low selection pressure makes the search equivalent to a random search process, reducing the ability to exploit promising regions of the search space over other regions [137]. A scarcely diversified population can cause a premature stagnation of the search, thus increasing the probability to be trapped in a local optimum. This phenomenon, called *premature convergence*, is a critical problem when dealing with GAs [306]. Such a phenomenon is observable in Genetic Algorithms (GAs) when the genetic operators (crossover and mutation) are unable to produce an offspring which outperforms its parents. This happens because individuals that have already been found during the search process can be regenerated again, while other parts of the search space can be left unexplored. The population diversity is crucial also to obtain a wider Pareto front in multi-objective problems. Indeed, the goals of a multi-objective optimization are (i) to guide the search towards the Pareto-optimal front and (ii) to maintain population diversity in trade-off fronts [106]. However, when applying

a GA for solving multi-objective problems, it is generally difficult to preserve the diversity of the individuals across different generations. Indeed, a multi-objective GA tends to create a limited number of groups of solutions (niches), which are close to each other in the search space, leaving the rest of the search space unexplored. Better diversified Pareto fronts will contain a wider range of solutions that, when recombined through the crossover operator, are more likely to lead towards a better achievement of the optimization objectives.

Selective pressure and population diversity are two conflicting factors [307]. Maintaining population diversity can affect the selective pressure, while increasing selective pressure can lead a faster loss of population diversity. Selective pressure can be guaranteed by traditional genetic operators, such as selection and crossover operation that allows to exploit the search space by looking for nearby better solutions. While mutation and random initialization are generally used to guarantee an adequate level of exploration. Mutation can help to reduce the population drift, i.e., if the mutation rate is high it will diversify the population, but at the same time it will prevent convergence and the solutions will remain at a certain distance to the optimum with higher probability [91]. For these reasons several methods have been proposed in order to promote diversity between the selected test cases by acting on different components of the GA, i.e., (i) by modifying the fitness function, (ii) by adding new similarity-based fitness functions, and (iii) by promoting diversity during the selection process. For example, widely used diversity-preserving mechanisms are *fitness sharing* [136, 144], *crowding distance* [308], *restricted tournament selection* [309], *rank scaling selection* [310].

Niching methods, based on the mechanism of *natural ecosystems*, is one of the first methods to deal with the population drift [136, 311]. For example, the *fitness sharing method* [136, 312] is used to force an individual to share its fitness with other individuals occupying the same niche. This means that each individual's fitness value is decreased by an amount proportional to the number of similar individuals in the population. Recently Della Cioppa *e*t al. [313, 314] provided a further investigation on the role of fitness sharing and its parameters on the performance of GAs. Another approach to reduce the loss of diversity is based on replacement strategies, which insert new individuals in the population by replacing similar individuals (*crowding method*). In the standard *crowding method* [311, 315], an offspring replaces the most similar individuals taken from a subpopulation of size *cf* (*crowding factor*) randomly drawn from the global population. Mahfoud [308] improved the standard crowding schema by introducing competition between children and parents of identical niches: each child replaces the nearest parent if it has higher fitness. The concept of crowding distance is also used by NSGA-II [107] when selecting individuals belonging to the same Pareto rank.

Harik *et al.* [309] proposed the Restricted Tournament Selection (RTS) to adapt standard tournament selection for multimodal optimization. RTS picks a candidate replacement individual that is closest to the new child from a random sample of *cf* individuals (as in standard crowding). After the determination of the closest individual to the candidate child, a competition is held based on fitness between the child and selected individual. The one having the best fitness is then selected for inclusion in the solution population. Rank scaling [144], instead, ranks the individuals according to their raw objective value. This avoids the

possibility of a small number of highly fitted individuals dominating the reproduction process. A further mechanism to maintain diversity consists of using multiple sub-population evolved in parallel. For example Jie *et al.* [316] suggest to use a dual-population single-objective GA to evolve in parallel two populations, the first one for exploiting the search space and second one to preserve diversity. Multi-population mechanisms have been also used with MOGAs. For example VEGA [143] uses different sub-population, each one for each objective function. In all the multi-population based GAs, the multiple sub-populations are evolved in parallel and periodically a migration schema is used to merge chromosomes from different sub-populations. The number and size of the sub-populations can be fixed and kept unchanged across generations [317] or can modified generation by generation [316]. However, the performance of these GAs are sensitive to the migration policy, migration rates and size of the sub-populations [316].

In the context of software testing, the problem of loss of diversity has been considered as an intrinsic problem of GAs and then the majority of previous research works used the same diversity preserving technique used by the evolutionary computation community. There have been research efforts aimed at increasing diversity among test cases following varying criteria for measuring similarity between test cases. One such approach [92] relies on sampling the program data-state, by tracing variables via intensive instrumentation, and looking for test cases that result in a different state. Another approach [93] for measuring diversity among test cases was proposed based on the notion of Information Distance. In order to address the problem of loss of diversity, Battiti *et al.* [318] proposed to check for the repetition of solutions and added this scheme to the Tabu Search algorithm. With the new Tabu scheme, called Reactive Tabu Search, an appropriate size of the tabu list is learned in an automated way by reacting to the occurrence of cycles. If a diversity loss is diagnosed, i.e., several individuals are similar (or identical) to each other, then the search is diversified by making a number of random moves proportional to the moving average of the cycle length.

The multi-objective GA used by Yoo and Harman [68, 103] (i.e., NSGA-II) is of particular interest in the context of test case diversity since it uses the concept of *crowding distance* to decide which solutions (set of selected test cases) have to be selected for the next generation (in this case the diversity is promoted during the selection process). In particular, the crowding distance measures how a solution (set of selected test cases) is far from the rest of the population [319]—according to the objective functions—and then giving to diversified solutions higher probability to survive. From the test suite optimization point of view, this means that having a particular coverage criterion, the crowding distance will promote set of test cases that are diversified according to the corresponding coverage criterion. NSGA-II also selects individuals on the basis of a rank scaling function—similar to the one used by Hemmati *et al.* [310]—where the ranks of the individuals are obtained according to the dominance ranking produced by the *non-dominated sorting algorithm*.

In our previous work [11] we proposed to add diversity as a further objective function in a multi-objective paradigm in order to preserve diversity. In particular, a density function has been added, with the aim of ensuring diversity between the individuals of the population

according to the statement coverage criterion. Results indicated the usefulness of diversity to improve multi-objective GAs. Hemmati *et al.* [291] proposed to use test case diversity as the unique objective function to be optimized[3], by using similarity measures between test cases to maximize diversity among the selected test cases. Diversity is calculated using a similarity (or dissimilarity) measure between pairs of test cases, according to a given encoding of test cases as a sets or sequences of elements, where the encoding is generated by applying a coverage criterion on a test model [320]. For example, using the UML state machine-based testing, a test case can be viewed a sequence of state identifiers, then two test cases differ from each other if they have two diverse sets/sequences of state identifiers. Once a measure of distance is defined, an optimization algorithm can be applied to select diversified test cases until the maximum coverage level is reached or a sufficient amount of test cases is selected. Hemmati *et al.* also provided a set of strategies to select test cases such as Adaptive Random Testing (ART) [281] and single-objective GAs. Among the different selection strategies, GAs turned out to be the most effective technique for *similarity-based test case selection* [291]. Further studies [310, 320] also confirmed that the diversity plays an important role to increase the ability to detect faults and evolutionary algorithms also turned out to be most efficient for such an objective function.

The main disadvantage of previous approaches for test case diversity [11, 291, 310, 320] is that they use a density/distance function according to a given coverage criterion. In a multi-objective paradigm where multiple coverage criteria can be used, such approaches require to add a diversity function for each coverage criterion, thus increasing the number of objective functions to be (near) optimized. As noted by Köppen *et al.* [321], the performance of MOGAs rapidly decreases for an increasing number of objective functions. Thus, it is preferable to promote diversity without adding further objective functions, while acting on other steps of the evolution process, such as the selection mechanism or the generation of new individuals.

5.5 Improving evolutionary testing by diversity injection

Existing approaches try to guarantee the population diversity considering information about each generation separately. Considering only the diversity of the current population might not guarantee the exploration of new unexplored regions, since several regions can be revisited numerous times. In this thesis we present a novel diversity preserving technique that introduces diversity by exploring areas of the search space which are different from those traversed so far, based on the direction through which the population as a whole has evolved. The movements of individuals across different generations —called evolution directions— are caught using SVD [322], which gives an estimate of where the search is going to. Thus,

[3]Hemmati *et al.* [291] considered diversity in a single-objective paradigm without taking into account further important test case selection criteria, such as the execution cost.

diversity is introduced by replacing the worst individuals (individuals with the lowest fitness values) with new individuals from orthogonal evolution directions that will explore new search regions. The formal definition of evolution directions and how to estimates such directions through SVD are reported in Section 6.2. Finally, the proposed strategy was integrated within the main loop of GAs and MOGAs as described in Section 6.2.2.

To provide a preliminary evidence of the benefits of the proposed SVD-based diversity preserving techniques, Chapter 6 reports the results of an empirical study conducted on numerical benchmark test problems (15 single-objective and 12 multi-objectives benchmark test problems) generally used by the evolutionary computation community to test search algorithms. After this preliminarily evaluation, the usefulness of SVD-based technique for single-objective GAs will be further investigated in the context of evolutionary test data generation in Chapter 7. Finally, Chapter 8 will apply the proposed technique in the context of multi-objective test suite optimization.

Chapter 6

Improving genetic algorithms through diversity injection

Contents

6.1 Introduction 140

6.2 Injecting Diversity using SVD 141

6.2.1 Estimating the evolution directions using SVD 141

6.2.2 Generating orthogonal individuals 144

6.2.3 Integrating SVD with genetic algorithms 146

6.2.4 SVD complexity 148

6.3 Empirical study on single-objective GAs 148

6.3.1 Data Collection and Analysis 149

6.3.2 Empirical Results 150

6.4 Empirical study on MOGAs 153

6.4.1 Data Collection and Analysis 154

6.4.2 GA Parameter Settings 155

6.4.3 Empirical Results 156

6.5 Conclusion 161

6.1 Introduction

The effectiveness of GAs can be seriously impacted by genetic drift, a phenomenon that takes place when a population (set of candidate test cases) is dominated by a small set of similar individuals that lead GAs to converge to a sub-optimal solution and to stagnate, without reaching the desired objective. Depending on the difficulty of the current search target and the type of selection scheme in use, it can happen that nearly all the candidate individuals in the search become too similar. Consequently, genetic operators simply recombine genetic material among these similar individuals and generate offsprings quite similar to their parents.

During recent and past years, different kinds of techniques have been proposed to deal with *population drift* (or *genetic drift*) [91, 306] in order to avoid premature convergence toward local optima. Several approaches have been proposed to improve the efficiency of GAs in terms of convergence rate and solution accuracy, such as parameters tuning [323, 324, 325], niching and fitness sharing [313, 314, 312, 136, 311], distance crowding [315, 107], multi-population algorithms [317, 316]. All these techniques aim at guaranteeing the population diversity considering information about each generation individually, without reducing the likelihood to regenerate individuals that have already been considered in the search process.

In this chapter, we propose the usage of a linear algebra technique, the Singular Value Decomposition (SVD) [322], to estimate the *evolution directions* of a population across different generations to promote the exploration of unexplored regions by creating new individuals with orthogonal evolution directions. In this context, for evolution direction we intend the direction along which the best individuals of a population are evolving in the genotype space across two different generations, independently from the objective space of the fitness function(s). Such an independence allows us to design a preserving diversity techniques that can be applied to both single-objective and multi-objective problems. This chapter gives the following contributions:

- We define the concept of evolution directions of a population across different generations, independently from the nature of the objective space (i.e., if there is only one or more than one objective to be optimized); such evolution directions are estimated through linear algebra theorems and SVD.

- We also describe a set of linear algebra operations to inject new individuals with orthogonal evolution directions thus increasing diversity during the search process. Such a diversity preserving mechanism is also integrated within the main loop of single-objective GAs and NSGA-II.

- We provide a preliminary evaluation of the usefulness of the enhanced single-objective GAs, named SVD-GA, by solving 15 single-objective benchmark test functions and comparing its performance with the standard GA with a distance-crowding schema and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [326, 327] which is one of the fastest algorithm for numerical problems.

- We evaluated the usefulness of the enhanced NSGA-II, named SVD-NSGA-II, by solving 12 multi-objectives benchmark test problems and comparing its performance with the original NSGA-II.

The chapter is organised as follows. Section 6.2 describes how to apply SVD to estimate the evolution directions and how generate orthogonal individuals. Section 6.2.2 describes how to integrated SVD-based diversity mechanisms into the main loop of GA and NSGA-II, presenting SVD-GA and SVD-NSGA-II. Experiments carried out on the proposed algorithms follow in Sections 6.3 and 6.3. The conclusions of this chapter can be found in Section 6.5.

6.2 Injecting Diversity using SVD

In this section we present a sophisticated diversity-preserving technique based on linear algebra operations to introduce diversity during the search process by periodically injecting new diversified individuals. Specifically, first we estimate the directions along which the population is evolving and then we use such directions to generate new orthogonal individuals. The idea is that injecting diversity the capability of GAs (either MOGAs) to escape from local optimal regions will increase avoiding population drift.

6.2.1 Estimating the evolution directions using SVD

Principles of linear algebra[1] can be used to analyse which part of the search space is explored by a population at a given generation t. Indeed, it can be possible to measure whether two or more individuals are exploring the same search space region by using the concept of orthogonality.

Definition 11. *Let us to consider for simplicity two solutions $x = \{x_i, \ldots, x_n\}$ and $y = \{y_i, \ldots, y_n\}$. The two solutions are said to be orthogonal (or linear independent) if and only if their inner product is equal to zero, i.e. if*

$$\langle x \cdot y \rangle = x_1 y_1 + \cdots + x_n y_n = 0$$

otherwise they are said to be linear dependent.

The lower the absolute value of the inner product, the higher the distance between x and y within the search space. Clearly, this analysis can be generalized for more than two single solutions: if all individuals in a given population are mutually orthogonal[2], this means that they are exploring different regions and there are not individuals that explore the same region. Vice versa, if all individuals are linear dependent to each other, then, they are exploring the same region of the search space (equivalently the maximum number of linear independent

[1] In this section we assume a familiarity with the basic terminology of linear algebra and refer the reader to [322] for a more complete coverage.

[2] A set of vectors V is mutually orthogonal if and only if all pairs of vector x and y in V are orthogonal. Formally $\forall x, y \in V, \langle x \cdot y \rangle = 0$

individuals is zero). Hence, the number of regions explored by a population is equals to the maximum number of linear independent individuals, or the maximum number of individuals mutually orthogonal. The directions along which the individuals show the most of variation indicate which directions or regions are actually explored.

All these information can be captured using matrix linear operations. The idea is that a population of solutions P obtained by a GA (or a multi-objective GA) at a given generation t can be viewed as an $m \times n$ matrix:

$$P_t = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{bmatrix}$$

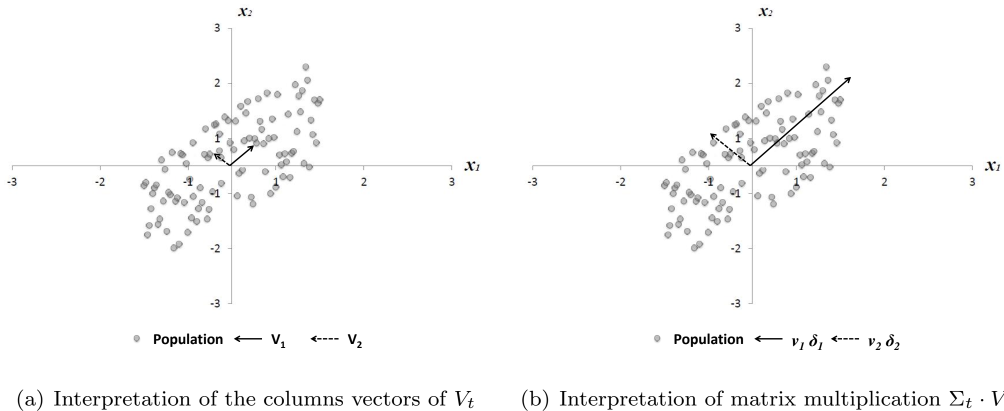
where n is the number of decision variables, m is the population size, while the generic entry $p_{i,j}$ denotes the value of the jth decision variable for the ith individual. The maximum number of linearly independent individuals (rows in P_t) corresponds to the rank of the matrix P_t, hence, the rank is equal to the number of explored directions. To derive which are such a directions we can use the Singular Value Decomposition (SVD). According to linear algebra theorem a rectangular matrix can always be factorized through SVD [328]:

Theorem 1. *Let P_t be an $m \times n$ matrix. Let r be the rank of P_t, then there exist (i) an $m \times r$ matrix U, (ii) an $r \times n$ matrix V and (iii) an $r \times r$ diagonal matrix Σ such that*

$$\begin{aligned} P_t &= U_t \times \Sigma_t \times V^T \\ &= \begin{bmatrix} u_{1,1} & \dots & u_{1,r} \\ \vdots & \ddots & \vdots \\ u_{m,1} & \dots & u_{m,r} \end{bmatrix} \times \begin{bmatrix} \sigma_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_r \end{bmatrix} \times \begin{bmatrix} v_{1,1} & \dots & v_{1,r} \\ \vdots & \ddots & \vdots \\ v_{n,1} & \dots & v_{n,r} \end{bmatrix}^T \end{aligned}$$

In this factorization the m rows of U_t are the *left singular vectors* while the n rows of V_t are *right singular vectors*. Finally, Σ_t is a $r \times r$ diagonal matrix whose diagonal entries are non negative and in descending order ($\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$). This decomposition contains several interesting information:

- The number of singular values r is the rank of the matrix P_t, hence, the number of singular values is equals to the number of directions explored by the individuals in P_t.
- The column vectors of V_t are the main directions along which the individuals show the most of variation. Thus, they measure which regions (or directions) are explored by the individuals in P_t.
- The diagonal elements of Σ represent the *importance* of each main direction of the population distribution.



(a) Interpretation of the columns vectors of V_t (b) Interpretation of matrix multiplication $\Sigma_t \cdot V_t$

Figure 6.1: Geometrical interpretation of SVD applied to a population P aiming at estimating evolution directions.

- Since the diagonal elements σ_i are in descending order, the corresponding vectors $V = \{v_1, v_2, \ldots, v_n\}$ are sorted according to their importance. Thus, the population exhibits the first largest variation along the direction v_1, the second largest variation along the direction v_2, and so on.

Figure 6.1-a shows the geometrical meaning of SVD decomposition for a population in a two-dimensional space, and the corresponding column vectors of V_t. It can be noted how the right singular values of V (i.e., the column vectors v_1 and v_2 in the figure) represent the main directions (eigenvectors) along which the population is distributed. It is also important to note that v_1 and v_2 are two orthonormal vectors, thus they represent only the main directions of P, without bringing information about how important they are. Such information is provided by the diagonal elements σ_1 and σ_2 in Σ, that measure the importance (eigenvalues) of the main directions for such a population (see Figure 6.1-b). As it can be seen in the example of Figure 6.1, the population exhibits the largest variation along v_1 ($\sigma_1 \geq \sigma_2$) confirming the theoretical definition of SVD. In summary, the SVD applied to a single population P_t allows to measure (i) how the individuals are distributed in the search space; (ii) whether more solutions are exploring the same directions of the search space; (iii) how many independent directions are actually explored.

Two different populations have two different SVD factorizations because they explore different directions of the search space. Thus, a population at generation t might have a different SVD factorization with respect to the directions of the same population considered at generation $t + k$. If the two SVD factorizations are identical then the population is unchanged across the k generations; otherwise, if the two factorizations are different this means that the population itself is changed to explore (to evolve towards) different directions of the search space. We defines this changes between generations t and $t + k$ as *evolution directions* and they measure the directions along which the population is evolved during these

Algorithm 6: ORTHOGONAL-VECTORS

Input:
A $m \times n$ matrix $\overline{V}$;
Result: A matrix $\overline{V}^o$;

```
1 begin
2   foreach Column i ∈ {1, ..., n} do
3     v_i^⊥ ⟵ reverse-order(v_i)
4     randomly multiply by -1 ⌊m/2⌋ of elements in v_i^⊥
5     if m mod 2 ≠ 0 then
6       randomly set to zero one of ⌈m/2⌉ unmodified element in v_i^⊥
```

k generations. The estimation of the evolution directions can be performed by comparing the two corresponding factorizations $\{\Sigma_t, V_t\}$ at generation t and $\{\Sigma_{t+k}, V_{t+k}\}$ at generation $t+k$.

Definition 12. *Let* $P_t = \left(U_t \cdot \Sigma_t \cdot V_t^T\right)$ *and* $P_{t+k} = \left(U_{t+k} \cdot \Sigma_{t+k} \cdot V_{t+k}^T\right)$ *be the two SVD factorizations of a population obtained at two different generations* t *and* $t+k$*. The evolution directions can be estimated through the following two matrices:*

$$\overline{V} = V_{t+k} - V_t \qquad \overline{\Sigma} = \Sigma_{t+k} - \Sigma_t$$

where $\overline{V}$ *estimates the direction along which the population is evolving, and* $\overline{\Sigma}$ *measures the magnitude of its evolution.*

In other words, $\overline{V}$ is a rotating factor because it measures how the population is rotated during the k generations, while $\overline{\Sigma}$ is a scaling factor because it measures how the population is rotated during the k generations. If both $\overline{V}$ and $\overline{\Sigma}$ are zero matrix it means that P_t and P_{t+k} are identical and the population is not changed. If $\overline{V} = 0$ and $\overline{\Sigma} \neq 0$, this means that P_{t+k} is not rotated with respect to P_t but its individuals are translated in the search space. Finally, if $\overline{V} \neq 0$ and $\overline{\Sigma} = 0$, this means that P_{t+k} is rotated with respect to P_t or, equivalently, that the individuals changed their trajectories during the last k generations.

6.2.2 Generating orthogonal individuals

The evolution directions defined in the previous section can be used to force the evolution along different directions by injecting new individuals with orthogonal evolution directions in the last population P_t. The steps needed to inject diversity during the evolution are reported in Algorithm 7. The algorithm takes as input two populations P_t and P_{t+k} obtained by a GA at generations t and $t+k$ and produce as output a new population P^* obtained by injecting diversity in population P_{t+k}. The first two steps of the algorithm (2)-(3) select 50% of the best individuals from the two input populations, $P_t^{'}$ and P'_{t+k} respectively. The selection of the best 50% of individuals is performed using the native selection operator used by the

Algorithm 7: SVD-ORTH-INJECT

Input:
A population P_t
A population P_{t+k}
Result: A new population P^*_{t+k}

1 **begin**
2 $P'_t \longleftarrow$ 50% of best individuals of P_t
3 $P'_{t+k} \longleftarrow$ 50% of best individuals of P_{t+k}
4 $[U_t, S_t, V_t] \longleftarrow svd(P'_t)$
5 $[U_{t+k}, S_{t+k}, V_{t+k}] \longleftarrow svd(P'_{t+k})$
6 $\overline{V} \longleftarrow V_{t+k} - V_t$ //Evolution directions
7 $\overline{\Sigma} \longleftarrow \Sigma_{t+k} - \Sigma_t$ //Shifting operator
8 //Generate orthogonal evolution directions
9 $\overline{V}^o \longleftarrow$ ORTHOGONAL-VECTORS($\overline{V}$)
10 //Generate new orthogonal individuals
11 $P^o \longleftarrow U_{t+k} \cdot \left(\Sigma_{t+k} + \overline{\Sigma}\right) \cdot \left(V_{t+k} + \overline{V}^o\right)^T$
12 Push unfeasible individuals in P^o back to the feasible reagion
13 $P^* \longleftarrow P'_{t+k} \cup P^o$

specific GA to select the best solution at each iteration of its main loop. For example, if we used NSGA-II, then the 50% of best individuals are selected using the *fast non-dominated sorting algorithm* and the *crowding distance* [107]. While with single-objective GAs the selection is performed through the tournament selection operator.

Hence, SVD is applied in steps (4)-(5) to decompose each of these two sub-populations as $P' = \left(U \cdot \Sigma \cdot V^T\right)$ for identifying and ordering the dimensions (axes) along which the individuals exhibit most of the variation. In steps (6)-(7) the algorithm computes the evolution directions according to Definition 12. Starting from $\overline{V}$ and $\overline{\Sigma}$, it is possible to generate a new set of individuals P^o as reported in line (11) with orthogonal directions as compared to directions of the current population. The factor $\left(\Sigma_{t+k} + \overline{\Sigma}\right)$ is the *shifting operator* that allows us to create new individuals that are $\overline{\Sigma}$-shifted in the search space, while the factor $(V_{t+k} + \overline{V}^o)$ is the *orthogonal operator* which creates the new individuals with orthogonal evolution directions, i.e., new individuals which explore new sub-spaces. The matrix $\overline{V}^o$ computed at line (9) is a matrix whose column vectors are unit and orthogonal with respect to column vectors of $\overline{V}$, i.e., $\overline{v}^o_i$ is orthogonal to $\overline{v}_i$ for $i = 1 \ldots n$. Since the number of all possible orthogonal column vectors is infinite, a good choice would be to randomly generate such vectors. We have chosen the simple method shown in Algorithm 6 [5]. Such an algorithm creates orthogonal vectors in a simple way. Given a vector $\overrightarrow{v}_i$, its orthogonal vector can be computed by (i) creating a new vector $\overrightarrow{v}^o_i$ with the same elements in $\overrightarrow{v}_i$ but in reverse order, and (ii) multiplying 50% of its elements by -1. If the number of elements in $\overrightarrow{v}^o_i$ is odd, then we randomly set to zero one of its unmodified element in order to ensure the

Algorithm 8: SVD-GA.

Input:
Number of variables N
Population size M
SVD interval k
Result: A solution S to the problem

```
1  begin
2      t ⟵ 0 // current generation
3      generate initial population P_t
4      old ⟵ t
5      while not (stop condition) do
6          //main loop of a GAs
7          generate offsprings Q_t using crossover and mutation
8          select P(t+1) from P(t) and Q(t)
9          t ⟵ t+1
10         if t mod k = 0 then
11             P_t ⟵ SVD-ORTH-INJECT(P_old, P_t)
12             old ⟵ t
13     S ⟵ best individual of P_t
```

orthogonality with $\vec{v}_i$ (lines 5-6 in Algorithm 6). The remaining issue to solve is that the orthogonal individuals generated using SVD might be violate one or more constraints of the optimization problem. To address this issue, we push these unfeasible individuals p^* back into the feasible region in line 11 of Algorithm 7.

In summary, applying Algorithm 7 we obtain a new population P^* that contains some new orthogonal individuals computed by SVD. As it can be noted such an algorithm works by comparing the sets of solutions obtained at two different generations, i.e., without modifying any evolutionary genetic operator or objectives to be optimized, as niched GA does. Moreover, the proposed SVD-based technique estimates the evolution direction within the search space or genotype, thus the use of SVD does not require further fitness function evaluations.

6.2.3 Integrating SVD with genetic algorithms

This subsection describes how to integrate the SVD-based mechanism into the main loop of single-objective GA and NSGA-II. Algorithm 8 outlines the pseudo-code of the single objective GA enhanced with the proposed diversity preserving mechanism, we refer to this algorithm as SVD-GA. First and foremost, in line 3 a randomly and uniformly distributed population P_t is created. Then, the main loop of the SVD-GA algorithm first includes k iterations of the original GA, i.e. lines 7-8. During these k iterations the usual selection, recombination, and mutation operators are used to create new solutions and to select the solutions that have to survive for the next iterations. Then, every k generations we apply the

Algorithm 9: SVD-NSGA-II

```
Input:
A test suite of size N
Population size M
SVD interval k
Result: A set of non-dominated solutions S
1  begin
2      t ⟵ 0 // current generation
3      P_t ⟵ RANDOM-POPULATION(N,M)
4      old ⟵ t
5      while not (end condition) do
6          Q_t ⟵ MAKE-NEW-POP(P_t)
7          R_t ⟵ P_t ⋃ Q_t
8          𝔽 ⟵ FAST-NONDOMINATED-SORT(R_t)
9          P_{t+1} ⟵ ∅
10         i ⟵ 1
11         while | P_{t+1} | + | 𝔽_i | ⩽ M do
12             CROWDING-DISTANCE-ASSIGNMENT(𝔽_i)
13             P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_i
14             i ⟵ i + 1
15         Sort(𝔽_i) //according to the crowding distance
16         P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_i[1 : (M− | P_{t+1} |)]
17         t ⟵ t + 1
18         if t mod k = 0 then
19             P_t ⟵ SVD-ORTH-INJECT(P_old, P_t)
20             old ⟵ t
21     S ⟵ P_t
```

SVD-based preserving technique on the past and current populations P_{old} and P_t respectively by invoking Algorithm 8. This algorithm requires to select the best 50% of individuals from the two populations. In this case the selection is performed through the tournament selection operator.

Algorithm 9 outlines the pseudo-code of the NSGA-II enhanced with the proposed diversity preserving mechanism, we refer to this algorithm as SVD-NSGA-II. The algorithm starts by generating a randomly and uniformly distributed population P_t. Then, the main loop of the SVD-GA algorithm first includes k iterations of the original GA, i.e. lines 7-8. During these k iterations the usual recombination, and mutation operators are used to create offsprings while the selection is applied by using the *fast non-dominated sorting algorithm* and the *crowding distance*. The difference with NSGA-II is that every k generations we apply the SVD-based preserving technique on the past and current populations P_{old} and P_t respectively. The selection of the best 50% of individuals in Algorithm 9 is performed using

the *fast non-dominated sorting algorithm* and the *crowding distance* [107]. Specifically, we add as many individuals as possible to the set of best individuals, according to their non-dominance ranks. If the number of best individuals is smaller than 50% of the population size $M/2$, then further individuals are selected from the next dominant rank according to the descending order of crowding distance.

6.2.4 SVD complexity

The SVD of a matrix P is typically computed in two steps [322]. In the first step, the matrix is reduced to a bi-diagonal matrix[3]. The computational complexity of this step is $\mathcal{O}(m \cdot n^2)$, assuming that $m > n$. In the second step, an iterative method is applied to compute the SVD of the bi-diagonal matrix. This method requires $\mathcal{O}(n)$ iterations, where each iteration has a complexity of $\mathcal{O}(n)$. Thus, the overall cost of SVD is $\mathcal{O}(m \cdot n^2)$ [328].

SVD is a factorization of a real or complex matrix, with many useful applications in different fields. In numerical analysis, the SVD provides a measure of the effective rank of a given matrix [322], while in statistics and time series analysis, the SVD is particularly useful tool for finding least-squares solutions and approximations [329]. The properties of SVD can be successfully used in image processing for compression and noise reduction [330], as well as in Information Retrieval to deal with linguistic ambiguity issues [78].

SVD has also been employed for characterizing protein molecular dynamics trajectories [331]. Similar to this work, we propose to integrate SVD in a GA to (i) estimate the evolution direction of a population and (ii) add to the population individuals with different characteristics, allowing the GA to explore many search sub-spaces in parallel.

6.3 Empirical study on single-objective GAs

This section reports the design and results of the numerical experiments we conducted to analyze the benefits of SVD-GA for single-objective problems. Specifically, we compared SVD-GA with the single-objective GA and with the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [326, 327]. The implementation of all the algorithms is based on the MATLAB's Global Optimization Toolbox [261]. SVD-GA is implemented by customizing the MATLAB's routine *ga* while the singular value decomposition is computed by using the routine *svd*. For the evolution strategy we use the MATLAB implementation of CMA-ES[4], restarted with increasing population size [332].

The three algorithms have been experimented on 8 multimodal and 7 unimodal benchmark functions [333, 334, 335, 22] reported in Table 6.1. For some of them the Matlab implementation is available online[5]. Functions $f_1 - f_8$ are multimodal functions where the

[3] A bi-diagonal matrix is a matrix with non-zero entries along the main diagonal and either the diagonal above or the diagonal below.

[4] http://www.lri.fr/∼hansen/cec2005.html

[5] http://www.maths.uq.edu.au/CEToolBox/node3.html

Table 6.1: Single-objective test functions used in our numerical experiment. The functions $f_7, f_8, f_9, f_{14}, f_{15}$ come from CEC 2008 Special Session [22].

Test Function	**n**	**Solution Space**	f_{min}
Rastrigin's function	50	$[-5.12; 5.12]^n$	0
Ackley's function	50	$[-32; 32]^n$	0
Griewangk's function	50	$[-600; 600]^n$	0
Generalized penalized function 1	50	$[-50; 50]^n$	0
Generalized penalized function 2	50	$[-50; 50]^n$	0
Shifted Rastrigin's function	50	$[-5; 5]^n$	-330
Shifted Ackley's function	50	$[-32; 32]^n$	-140
Shifted Griewangk's function	50	$[-600; 600]^n$	-180
Sphere model	50	$[-100; 100]^n$	0
Schwefel's Problem 2.22	50	$[-10; 10]^n$	0
Schwefel's Problem 2.21	50	$[-100; 100]^n$	0
Schwefel's Problem 1.2	50	$[-100; 100]^n$	0
Schwefel's sin root function	50	$[-512; 512]^n$	-418.98
Shifted Sphere model	50	$[-100; 100]^n$	-450
Shifted Schwefel's Problem 2.21	50	$[-100; 100]^n$	-450

number of local minima increases exponentially with the problem dimension (i.e., the number of independent variables). Instead, functions $f_9 - f_{15}$ are high-dimensional unimodal problems that, because of their convexity, cause poor or slow convergence of the optimization algorithms toward a single global minimum. The functions $f_6 - f_8$ and the functions $f_{14} - f_{15}$ are shifted benchmark functions provided for the CEC 2008 Special Session and Competition on Large Scale Global Optimization [22]. All the benchmark functions considered in our case study are deemed to be representative of difficult classes of problems for many optimization algorithms (including evolutionary algorithms) [333, 334, 335]. It is worth noting that for unimodal functions, it is more useful to analyze the convergence rates than the final results of fitness function. Instead, for multimodal function, the final results are much more important since they represent the ability of an optimization algorithm to escape from local optima and moves toward (near) global optimum. For all benchmark functions we set the problem dimension (n) to 50 and 100 in order to increase the difficulty in obtaining the global optimum. Note that previous works usually set such a number to 30 optimum [333, 334, 335].

6.3.1 Data Collection and Analysis

We run the three algorithms 50 times on each test function to address their randomness natures. Since optimal values of all the test functions are known, for each run, we stored the best (minimum) function error values $(f(x) - f^*(x))$ at varying number function evaluations (5×10^2, 5×10^3, 5×10^4, 5×10^5 respectively) similarly as done in [22]. Once all the data are collected, we computed the descriptive statistics of the best function error values achieved in

each run, i.e., median, mean, and standard deviation of the best function error values.

We also statistically compare the results achieved by SVD-GA and the other two algorithms on each test functions. The statistical analysis is used to test the following null hypothesis:

> *There is no difference between the error values achieved by SVD-GA and GA (or CMA-ES) when solving the test problem f_i.*

where $i = 1 \dots 15$. To compare samples from two data sets (e.g., the best values achieved in each run by the SVD-GA and GA) and test our hypothesis, we use the Wilcoxon Rank Sum test [226].

In all our statistical tests we consider p-values < 0.05 as statistically significant, i.e., we accept a 5% chance of rejecting a null hypothesis when it is true [226]. However, since three pairwise tests are necessary (simple GA vs SVD-GA, GA vs. CMA-ES, and SVD-GA vs. CMA-ES), we reject the null hypothesis if all p-values are smaller than $\frac{0.05}{3} = 0.016$, i.e., we correct p-values using the Bonferroni correction [226].

For the experimented GAs, we adopted the following parameter settings or scheme as done in previous studies (for example [336, 337]):

- **Population size**: we choose a moderate population size. Specifically, we set $p = 100$.
- **Initial population**: for each of test function the initial population is uniformly and randomly generated within the solution spaces reported in Table 6.1.
- **Selection**: we used the tournament selection operator as default for the MATLAB implementation.
- **Crossover operator**: we use the arithmetic crossover [261] with probability $P_c = 0.60$.
- **Mutation operator**: we use a uniform mutation operator with probability $P_m = 1/n$ where n is the number of independent variables such that, on an average, one variable gets mutated per individual [337].
- **Stopping criterion**: if the fitness value of the best individual cannot be further optimized in the subsequent 50 generations, then the execution of the GA is stopped. In addition, we set the maximum number of function evaluations equals to 500,000.
- **Elitism**: the number of individuals in the current generation that are guaranteed to survive to is 2.

6.3.2 Empirical Results

Table 6.2 reports the descriptive statistics of the error values achieved by GA, CMA-ES and SVD-GA for multimodal test functions when the number of independent variables was set to $n = 50$ (results obtained for $n = 100$ are reported in Appendix A). Comparing the performances of SVD-GA with GA, we can note that the proposed diversity preserving technique

Table 6.2: Function error values achieved on multimodal test functions by GA, SVD-GA, and CMA-ES when n=50. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference.

f	Eval.	GA			SVD-GA			CMA-ES		
		Median	Mean	St. Dev.	Median	Mean	St. Dev.	Median	Mean	St. Dev.
f1	5×10^2	3.65e+2	3.67e+2	3.00e+1	**1.45**	**6.83e+1**	1.13e+2	1.03e+3	1.03e+3	1.29e+2
	5×10^3	9.82e+2	9.97e+1	1.33e+1	**0**	**8.40**	1.70e-1	3.14e+2	3.27e+2	5.43e+1
	5×10^4	1.44e+1	1.48e+1	3.46	**0**	**0**	0	4.98e+1	5.65e+1	2.12e+1
	5×10^5	2.81e-2	6.09e-1	8.17e-1	**0**	**0**	0	2.98	3.66	1.45
f2	5×10^2	1.52e+1	1.51e+1	7.40e-1	**3.20e-1**	**5.91e-1**	7.47e-1	2.16e+1	2.16e+1	8.23e-2
	5×10^3	5.94	6.06	7.90e-1	**1.51e-14**	**3.19e-14**	4.01e-14	2.16e+1	2.16e+1	6.05e-2
	5×10^4	4.78e-2	5.38e-2	1.70e-2	**4.44e-15**	**4.87e-15**	3.13e-15	2.16e+1	2.16e+1	8.92e-2
	5×10^5	4.78e-2	5.37e-2	1.69e-2	**4.44e-15**	**4.87e-15**	3.13e-15	2.00e+1	2.07e+1	8.04e-1
f3	5×10^2	1.38e+2	1.40e+2	2.84e+1	**2.06e-1**	**3.78e-1**	3.44e-1	6.40e+2	6.36e+2	1.67e+2
	5×10^3	4.62	4.94	1.34	**0**	**0**	0	2.73e-3	3.36e-3	3.54e-3
	5×10^4	3.56e-2	3.79e-2	1.17e-2	**0**	**0**	0	4.44e-16	4.79e-16	3.11e-16
	5×10^5	1.25e-4	5.89e-3	6.77e-3	**0**	**0**	0	3.33e-16	3.64e-16	2.55e-16
f4	5×10^2	2.67e+6	3.25e+6	2.12e+6	**7.02e-1**	**7.46e-1**	1.73e-1	1.28e+1	6.66e+5	3.24e+6
	5×10^3	1.15e+1	1.17e+1	3.80	**1.52e-3**	**1.69e-3**	6.49e-4	7.61	2.60e+1	9.03e+1
	5×10^4	1.07e-4	5.35e-4	1.49e-3	**1.95e-23**	**2.90e-23**	2.34e-23	1.18e-5	9.95e-1	1.48
	5×10^5	6.20e-8	6.05e-8	2.06e-8	**9.42e-33**	**9.42e-33**	1.40e-48	3.15e-17	2.34e-16	6.34e-16
f5	5×10^2	1.38e+7	1.68e+7	1.18e+7	**5.88**	**6.09**	1.32	4.54e+1	4.68e+5	1.41e+6
	5×10^3	1.06e+2	1.58e+2	2.39e+2	**9.29e-2**	**9.41e-2**	4.06e-2	4.20e-1	1.09e+5	3.79e+5
	5×10^4	1.61e-3	3.14e-3	3.77e-3	**7.05e-22**	**1.35e-21**	2.44e-21	6.16e-7	3.17e-3	7.22-3
	5×10^5	2.15e-6	2.29e-6	1.32e-6	**1.35e-32**	**1.35e-32**	5.59e-48	1.21e-17	4.32e-17	9.80e-17
f6	5×10^2	6.48e+2	6.50e+2	4.43e+1	**6.40e+2**	**6.42e+2**	4.07e+1	9.80e+2	9.88e+2	1.28e+2
	5×10^3	1.53e+2	1.55e+2	2.36e+1	**1.03e+2**	**1.05e+2**	1.72e+1	3.50e+2	3.59e+2	6.43e+1
	5×10^4	1.07e+1	1.09e+1	2.58	**9.99e-1**	**9.79e-1**	8.90-1	7.06e+1	6.79e+1	1.98e+1
	5×10^5	1.90e-3	2.92e-1	5.35e-1	**6.12e-8**	**1.17e-7**	1.69e-7	3.98	3.93	1.97
f7	5×10^2	**2.00e+1**	**2.00e+1**	2.64e-1	2.01e+1	**2.00e+1**	8.58	2.16e+1	2.16e+1	7.30e-2
	5×10^3	**8.15**	**8.15**	9.23e-1	8.22	8.18	1.70e+1	2.16e+1	2.16e+1	6.31e-2
	5×10^4	3.34e-1	6.01e-1	5.33e-1	**4.86e-3**	**4.94e-3**	5.70e+1	2.16e+1	2.14e+1	4.42e-1
	5×10^5	2.57e-3	2.71e-3	6.52e-4	**2.91e-5**	**2.83e-5**	0	2.00e+1	2.06e+1	7.73e-1
f8	5×10^2	6.93e+2	7.06e+2	8.95e+1	**9.93e-1**	**8.29e-1**	3.89e-1	6.33e+2	7.00e+2	3.13e+2
	5×10^3	1.28e+1	1.31e+1	3.44	**0**	**1.36e-14**	2.03e-14	1.87e-3	3.17e-3	4.14e-3
	5×10^4	8.86e-2	9.65e-2	2.98e-2	**0**	**3.41e-15**	9.43e-15	2.84e-14	2.84e-14	0
	5×10^5	4.15e-4	6.07e-3	8.37e-3	**0**	**0**	0	**0**	**0**	0

turns out to be very useful for multimodal problems having many local optima. Indeed, the results show that SVD-GA provides error values that are several orders of magnitude lower than those obtained by the standard GA, demonstrating a better ability to explore the search space with notable lower probability to be trapped in local optima. Moreover, SVD-GA also demonstrates its superiority over CMA-ES. Indeed, on 7 out of 8 multi-modal functions, SVD-GA achieved a median/mean error value several order of magnitude better the those achieved with CMA-ES. Particularly interesting is the difference—in terms of error values—between SVD-GA and CMA-ES for the Ackley's function (f_2) and its shifted version f_7 with global optimum on bounds. Such a function has an exponential number of local minima and only one global optimum. For all the independent runs, CMA-ES was trapped in a local optimum (with $f(x) \approx 21$) after few function evaluations. Even if the algorithm was restarted with an incremental population size, it still quickly converged toward the same

Table 6.3: Function error values achieved on unimodal test functions by GA, SVD-GA, and CMA-ES when n=50. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference.

f	Eval.	GA			SVD-GA			CMA-ES		
		Median	Mean	St. Dev.	Median	Mean	St. Dev.	Median	Mean	St. Dev.
f9	5×10^2	1.45e+4	1.48e+4	2.84e+3	**8.52e-1**	**3.39**	5.33	6.28e+4	6.65e+4	2.41e+4
	5×10^3	3.84e+2	4.00e+2	1.11e+2	**2.88e-24**	**1.87e-22**	5.47e-22	1.04e-4	1.68e-4	1.39e-4
	5×10^4	1.88e-2	1.94e-2	5.11e-3	**1.24e-156**	**3.35e-147**	1.29e-146	5.49e-16	5.86e-16	1.87e-16
	5×10^5	3.82e-5	4.29e-5	1.30e-5	**0**	**0**	0	5.30e-16	5.62e-16	1.85e-16
f10	5×10^2	7.14e+1	7.05e+1	7.18	*1.07*	**1.39**	3.27e-1	4.80e+41	3.51e+46	1.27e+47
	5×10^3	9.85	9.33	2.07	**1.77e-12**	**4.10e-11**	6.35e-12	3.25e+32	6.33e+43	2.45e+44
	5×10^4	7.27e-2	7.57e-2	1.25e-2	**1.00e-75**	**6.24e-71**	1.82e-75	9.98e+2	9.99e+2	2.32e+2
	5×10^5	4.47e-3	4.42e-3	1.32e-3	**8.79e-256**	**3.64e-253**	0	7.76e-11	8.40e-11	1.97e-11
f11	5×10^2	2.82e+4	3.02e+4	9.73e+3	4.91e+2	1.03e+4	2.10e+4	**1.58e+1**	**6.02e+1**	1.01e+2
	5×10^3	6.52e+3	6.88e+3	2.11e+3	**3.58e-2**	1.34e+1	4.17e+1	3.27	**1.01e+1**	1.84e+1
	5×10^4	2.43e+2	2.64e+2	7.48e+1	**2.58e-38**	**8.91e-28**	4.42e-27	8.59e-8	1.18e-3	3.70e-3
	5×10^5	3.33	3.52	1.15	**7.92e-157**	**1.49e-133**	7.33e-133	4.20e-17	4.21e-17	5.53e-17
f12	5×10^2	4.22e+1	4.22e+1	4.62	**6.91e-1**	**1.24**	1.80	2.63e+2	2.70e+2	4.34e+1
	5×10^3	2.54e+1	2.56e+1	3.87	**3.56e-10**	**5.69e-4**	2.03e-3	5.84e+1	6.42e+1	3.06e+1
	5×10^4	2.23	2.42	5.69e-1	**4.12e-34**	**1.48e-15**	7.40e-15	1.98e-1	3.14e-1	2.63e-1
	5×10^5	9.04e-2	9.31e-2	2.09e-2	**2.30e-76**	**1.02e-39**	5.10e-39	1.66e-2	3.43e-2	7.02e-2
f13	5×10^2	2.65e+2	2.64e+2	1.69e+1	**2.61e+2**	**2.61e+2**	1.51e+1	5.08e+3	5.48e+3	2.90e+3
	5×10^3	8.35e+1	8.48e+1	1.30e+1	**6.28e+1**	**6.22e+1**	8.64	3.00e+2	3.02e+2	1.03e+1
	5×10^4	1.41e-1	9.82e-1	1.52	**9.09e-13**	**1.78e-12**	3.22e-12	2.96e+2	2.96e+2	6.83
	5×10^5	5.15e-5	5.28e-5	2.20e-5	**7.39e-13**	**6.91e-13**	9.24e-14	2.97e+2	2.97e+2	4.24
f14	5×10^2	9.05e+4	8.93e+4	7.89e+3	8.55e+4	8.68e+4	1.14e+4	**7.73e+4**	**8.66e+4**	3.31e+4
	5×10^3	1.31e+3	1.37e+3	2.89e+2	6.04e+2	6.48e+2	2.08e+2	**1.50e-4**	**2.10e-4**	1.92e-4
	5×10^4	7.78e-2	8.39e-2	2.70e-2	4.61e-4	4.70e-4	2.29e-4	**5.68e-14**	**5.91e-14**	1.14e-14
	5×10^5	1.84e-4	1.91e-4	4.85e-5	2.84e-8	3.06e-8	1.44e-8	**0**	**0**	0
f15	5×10^2	3.34e+6	4.20e+6	2.97e+6	9.92e+5	1.79e+6	1.74e+6	**3.06e+1**	**1.04e+2**	2.24e+2
	5×10^3	2.63e+4	2.70e+4	6.11e+3	2.51e+4	2.54e+4	6.73e+3	**1.53e+1**	**2.39e+2**	8.12e+2
	5×10^4	1.22e+3	1.35e+3	4.01e+2	5.25e+2	5.60e+2	1.38e+2	**7.47e-9**	**2.08e-4**	1.00e-3
	5×10^5	2.18e+1	2.45e+1	9.09	9.26e-1	1.07	4.74e-1	**0**	**0**	0

local optimum. Similarly, GA was trapped in a local optimum (with $f(x) \approx 4.78 \cdot 10^{-2}$) after several function evaluations. Differently from the other algorithms, SVD-GA is able to cross the valley among the local optima and reaches near optimal solutions. A similar scenario occurs with the Rastrigin's function (f_1 and its shifted version f_6), which is a non-linear multimodal function. Also in this case CMA-ES turned out to be unable to converge to the global optimum, being trapped in a local optimum (in this case the restarting strategy with incremental population is still inefficacious), while SVD-GA achieved near-optimal solutions. Thus, from the comparison of all the algorithms, SVD-GA is the best one (consistent results have been also obtained with $n = 100$ as reported in Appendix A).

Table 6.3 shows the descriptive statistics of the error values achieved by GA, CMA-ES, and SVD-GA for unimodal test functions (functions $f_9 - f_{15}$) when the number of independent variables was set to n = 50. We can observe that SVD-GA achieves error values that are several orders of magnitude lower than those obtained by GA with the same number of function evaluations. A particular case is represented by the function f_{12}, that is a convex unimodal function. For such a function, the convergence speed of GA is very low. In particular, after 10^4 function evaluations, the mean error value achieved by SVD-GA is

lower than 10^{-2}, while GA provides mean error values greater than 30. Comparing CMA-ES and SVD-GA, the achieved results reveals that there is no a clear winner among them. For functions f_9-f_{13} SVD-GA obtains better performances when the number of function evaluations is high. Similar considerations can done for other functions—such as f_{10} and f_{13}. Vice versa, for the functions f_{14} and f_{15}, the winner algorithm is CMA-ES, that is able to converge towards the unique global optimum. In addition, CMA-ES is much faster than SVD-GA. This is not surprising, since for CMA-ES the adaptation follows a natural gradient approximation of the expected fitness [326, 327], while SVD-GA does not use any approximation of it. However, for unimodal functions SVD-GA demonstrated to be able to compete with CMA-ES, and to be sometimes faster than it. Consistent results have been also obtained with n = 100 as reported in Appendix A.

In summary, *SVD-GA is able to significantly improve both convergence speed and optimality of GA for both multimodal and unimodal functions. Moreover, SVD-GA outperforms the restarted CMA-ES with increasing population for multimodal functions, demonstrating its ability to escape from local optima, because of the diversity introduced by orthogonal sub-populations. Finally, SVD-GA turned out to be competitive with CMA-ES for highly convex unimodal functions.*

6.4 Empirical study on MOGAs

For the multi-objective problems, we compared the multi-objective SVD-NSGA-II with the NSGA-II as defined by Deb *et al.* [107]. The implementation of all the algorithms is based on the MATLAB's Global Optimization Toolbox [261]. For the NSGA-II algorithm, we used the routine *gamultiobj*, while parameter values (see sub-section 6.4.2) are set using *gaoptimset*. Also, the multi-objective SVD-NSGA-II is implemented in MATLAB by customizing the routine *gamultiobj*.

For the numerical experimentation, we selected 7 two-objectives and 8 three-objectives benchmark problems reported in Tables 6.4 and 6.5, respectively. Problems MOP1 and MOP2 are well-known two-objective problems introduced by Schaffer [143] and by Fonseca [338]. Since the optimum x^* for the first objective f_1 is not the optimum for the second objective f_2 and vice versa, the Pareto optimal set consists of more than one solution, including the solution that minimize each objective individually. Functions S_ZDT1, S_ZDT2, S_ZDT4 and S_ZDT6 are shifted benchmark functions of the well-known test suite ZDT [339] provided for the CEC 2007 Special Session and Competition on Real Parameter Multi-objective Optimization [340]. For these problems, the global optimum has different parameter values for different variables/dimensions and the global optimum is not located in the centre of the search space (i.e., the problems are not symmetric). Since from the competition there is no shifted formulation of the ZDT3 problem, we used its traditional formulation [339]. In addition, since all the selected benchmark problems are scalable to any number of decision variables, we set the problem dimension (n) to 50 and 100 in order to evaluate the performance of the different algorithms with increasing difficulty in converging to the real Pareto

Table 6.4: Two-objectives test problems used in our numerical experiment. The number of variables is $n = 50,\ 100$.

Test Function	**Solution Space**
Schaffer's Problem (MOP1)	$[-10^5; 10^5]$
Fonseca Problem (MOP2)	$[-2; 2]$
Shifted ZDT1	ref. [340]
Shifted ZDT2	ref. [340]
ZDT3	$[0; 1]^2$
Shifted ZDT4	ref. [340]
Shifted ZDT6	ref. [340]

Table 6.5: Three-objectives test problems used in our numerical experiment.

Test Function	N. Variables	Solution Space
DTLZ1	10, 20	$[0; 1]^n$
DTLZ2	50, 100	$[0; 1]^2 \times [-100; 100]^{n-2}$
DTLZ3	10, 20	$[0; 1]^n$
DTLZ4	50, 100	$[0; 1]^2 \times [-10; 10]^{n-2}$
DTLZ6	50, 100	$[0; 1]^2 \times [0; 10]^{n-2}$

front.

The three-objectives problems, DTLZ1-DTLZ4 and DTLZ6 are selected from the DTLZ [341] test suite. Such problems are widely adopted to test the ability of evolutionary algorithms to control both (i) the convergence speed toward the true Pareto-optimal front and (ii) maintaining a widely distributed set of solutions. For all problems, we used the three-objective formulations and two different problem dimensions as reported in Table 6.5.

6.4.1 Data Collection and Analysis

To analyse the performance of the various algorithms we used the Inverse Generational Distance (IGD) metric. Formally, let P^* be the optimal Pareto front, and let P be the approximation of the optimal Pareto front obtained by an evolutionary algorithm, the IGD measure can be defined as follows:

$$IGD(P, P^*) = \frac{\sum_{v \in P^*} dist(v, P)}{\mid P^* \mid}$$

where $dist(v, P)$ denote the minimum Euclidian distance between the optimal point $v \in P^*$ and the approximated front P. In this way, the IGD is able to measure both convergence (proximity to the optimal/real Pareto front) and diversity of the approximated front P. We run the two experimented algorithms (i.e., NSGA-II and SVD-NSGA-II) 50 times on each test function. For each run, we stored the IGD values achieved after 5×10^2, 5×10^3, 5×10^4, 5×10^5 functions evaluations, as done in [22]. Once collected all data, we computed the

descriptive statistics (i.e., median, mean, and standard deviation) of the IGD values achieved in each run. The IGD metric can be easily computed since for all the objective functions the true optimal is known a priori. However, since the Pareto optimal front contain an infinite number of solutions, we approximated the fronts by selecting a finite number of solutions uniformly distributed along the corresponding front. For functions S_ZDT1-S_ZDT6 we used the Pareto frontier provided by Huang [340], while for MOP1 and MOP2 we approximated the real Pareto frontiers with 200 distinct solutions uniformly distributed along the fronts. Finally, since the DTLZ problems have three objectives to be optimized, we approximated the corresponding true Pareto fronts with 1,000 distinct uniformly-distributed solutions.

We also statistically compare the results achieved by SVD-NSGA-II and NSGA-II on each test functions. The statistical analysis is used to test the following null hypothesis:

> *There is no difference between the best values achieved by SVD-NSGA-II and NSGA-II when solving the test problem f.*

To statistically compare the IGD values obtained by the two algorithms and to test our hypotheses, we use the Wilcoxon Rank Sum test [226]. In all our statistical tests we consider p-values < 0.05 as statistically significant, i.e., we accept a 5% chance of rejecting a null hypothesis when it is true [226].

6.4.2 GA Parameter Settings

For both NSGA-II and SVG-NSGA-II we adopted the following parameter settings or scheme as done in previous studies (for example [107, 337]):

- **Population size**: we choose a large population size $P = 200$ since we are interested in achieving a wide set of Pareto optimal solutions.
- **Initial population**: for each test problem, the initial population is uniformly and randomly generated within the solution spaces reported in Table 6.4.
- **Selection**: we used the tournament selection operator with tournament size equals to 2 as default for the MATLAB implementation of NSGA-II.
- **Crossover operator**: we use the arithmetic crossover [261] with probability $P_c = 0.60$.
- **Mutation operator**: we use a uniform mutation operator with probability $P_m = 1/n$ where n is the number of independent variables [107].
- **Stopping criterion**: if the average spread of non-dominated solutions cannot be further optimized in the subsequent 50 generations, then the execution of the GA is stopped [261].
- **Pareto fraction**: the percentage of solutions used for approximating the Pareto front (the set of non dominated solutions at each generation) was set to $P_f = 0.5$.

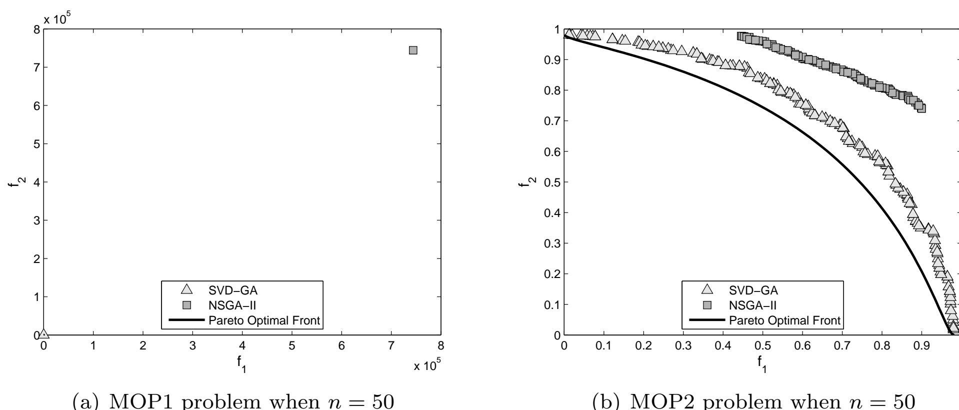
Table 6.6: IGD values achieved by SVD-NSGA-II and NSGA-II for n=50. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference.

f	Func. Eval.	NSGA-II			SVD-GA		
		Median	Mean	St. Dev.	Median	Mean	St. Dev
MOP1	$2 \cdot 10^5$	2.80e+3	6.54e+4	1.20e+5	**1.28e-1**	**1.42e-1**	3.99e-2
	$3 \cdot 10^5$	2.80e+3	6.54e+4	1.20e+5	**1.04e-1**	**1.04e-1**	4.18e-3
	$4 \cdot 10^5$	2.79e+3	6.54e+4	1.20e+5	**1.01e-1**	**1.01e-1**	5.77e-3
	$5 \cdot 10^5$	2.78e+3	6.54e+5	1.20e+5	**9.64-1**	**9.66e-2**	8.23e-3
MOP2	$2 \cdot 10^5$	5.83e-1	5.54e-1	9.28e-2	**5.23e-2**	**5.36-2**	3.67e-3
	$3 \cdot 10^5$	5.56e-1	5.09e-1	1.08e-1	**4.60e-2**	**4.66e-2**	1.87e-3
	$4 \cdot 10^5$	5.10e-1	4.58e-1	1.12e-1	**4.60e-2**	**4.59e-2**	1.50e-3
	$5 \cdot 10^5$	4.57e-1	4.08e-1	1.13e-1	**4.52e-2**	**4.47e-2**	1.91e-3
Shifted ZDT1	$2 \cdot 10^5$	3.10e-2	3.59e-2	1.58e-2	**1.39e-2**	**1.50e-2**	5.24e-2
	$3 \cdot 10^5$	2.153-2	2.54e-2	1.25e-2	**7.67e-3**	**7.89e-3**	9.49e-4
	$4 \cdot 10^5$	1.923-2	2.30e-2	1.16e-2	**6.88e-3**	**6.88e-3**	3.65e-4
	$5 \cdot 10^5$	1.813-2	2.20e-2	1.14e-2	**4.21e-3**	**4.39e-3**	4.70e-4
Shifted ZDT2	$2 \cdot 10^5$	2.54e-1	3.14e-1	2.64e-1	**1.91e-2**	**2.47e-2**	1.45e-2
	$3 \cdot 10^5$	2.38e-1	3.04e-1	2.69e-1	**9.43e-3**	**1.00e-3**	2.13e-3
	$4 \cdot 10^5$	2.36e-1	3.02e-1	2.70e-1	**7.47e-3**	**7.70e-3**	8.42e-4
	$5 \cdot 10^5$	2.35e-1	3.02e-1	2.71e-1	**5.03e-3**	**5.21e-3**	8.04e-4
ZDT3	$2 \cdot 10^5$	2.17e-2	2.59e-2	1.17e-2	**1.15e-3**	**1.13e-3**	2.40e-4
	$3 \cdot 10^5$	1.76e-2	2.21e-2	1.08e-2	**1.07e-3**	**1.07e-3**	1.83e-4
	$4 \cdot 10^5$	1.68e-2	2.11e-2	1.06e-2	**1.06e-3**	**1.05e-3**	1.81e-4
	$5 \cdot 10^5$	1.65e-2	2.07e-2	1.04e-2	**1.06e-3**	**1.02e-3**	1.58e-4
Shifted ZDT4	$2 \cdot 10^5$	6.93e+1	7.47e+1	1.14e+1	**4.46e+1**	**5.86e+1**	4.82e+1
	$3 \cdot 10^5$	3.87e+1	4.02e+1	1.01e+1	**1.24e+1**	**2.82e+1**	6.55e+1
	$4 \cdot 10^5$	2.46e+1	2.39e+1	6.64e	**5.32**	**6.51**	3.92
	$5 \cdot 10^5$	1.85e+1	1.93e+1	4.45	**1.88**	**2.79**	2.24
Shifted ZDT6	$2 \cdot 10^5$	1.29e-1	1.70e-1	1.52e-1	**6.79e-2**	**7.30e-2**	2.24e-2
	$3 \cdot 10^5$	4.67e-2	1.08e-1	1.33e-1	**1.51e-2**	**1.57e-2**	2.61e-3
	$4 \cdot 10^5$	3.13e-2	9.84e-2	1.34e-1	**1.03e-2**	**1.03e-2**	9.12e-4
	$5 \cdot 10^5$	3.12e-2	9.77e-2	1.35e-1	**8.83e-3**	**8.91e-3**	5.22e-4

6.4.3 Empirical Results

Table 6.6 reports mean, median and standard deviation of the IGD metrics achieved using NSGA-II and multi-objective SVD-NSGA-II for the two-objective test problems with problem size $n = 50$ (the results for $n = 100$ are reported in Appendix A). From the analysis of the results, we can note that SVD-NSGA-II is able to converge faster toward the real Pareto fronts for all the two-objective problems. Indeed, median, mean and standard deviation of IGD values achieved by SVD-NSGA-II are always smaller than those of NSGA-II. Moreover, the Wilcoxon rank sum test reveals that such differences are statistically significant.

To provide a visual comparison, we also show examples of simulation results when using NSGA-II and SVD-NSGA-II for some of the two-objective test problems. Figure 6.2-a shows the non-dominated solutions obtained by the two algorithms after $5 \cdot 10^5$ function evaluation on the MOP1 test problem. Such a problem has a huge search space, while the Pareto optimal front is located in a small region of the whole search space. This problem is usually used for measuring the speed of an evolutionary algorithm to converge to the Pareto optimal



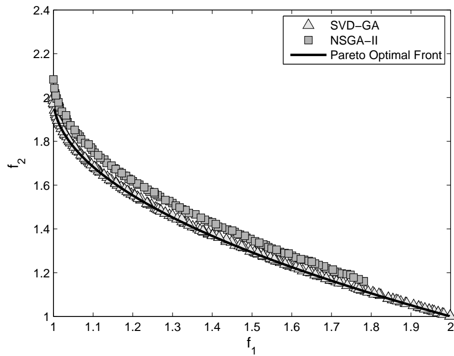
(a) MOP1 problem when $n = 50$

(b) MOP2 problem when $n = 50$

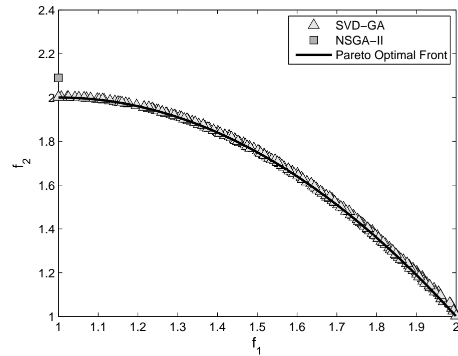
Figure 6.2: NSGA-II vs. SVD-NSGA-II on MOP test functions

front. The Figure clearly demonstrates the abilities of SVD-NSGA-II in converging to the true front, while NSGA-II is quite far from the Pareto-optimal region. Figure 6.2-b shows the non-dominated solutions obtained by the two algorithms, as well as the corresponding Pareto optimal front, for the MOP2 problem. This problem has a single non-convex Pareto-optimal front with a large and non-linear trade-off curve, which poses challenge to find and maintain the entire front uniformly distributed. SVD-NSGA-II is better than NSGA-II in terms of convergence and distribution of the solutions, demonstrating to be less affected by the exponential factor of the two objectives, which increases the difficulty to converge toward the real Pareto front.

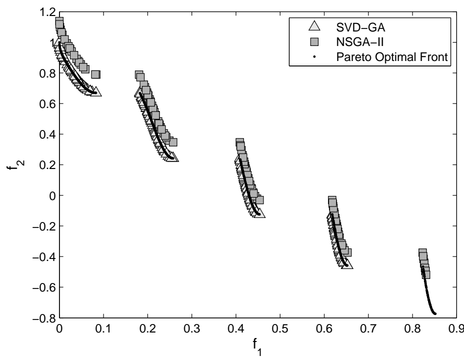
Figure 6.3-a shows the Pareto fronts obtained by the two algorithms on the shifted SZDT1 test problem. As we can see, NSGA-II was not able to maintain enough diversified non-dominated solutions in the final population and the distribution of the solutions is worst if compared to that obtained in the final population of SVD-NSGA-II. Figure 6.3-b shows the results achieved on the shifted ZDT2 test problem, which has a non-convex optimal Pareto front. We can observe that SVD-NSGA-II produced a front of solutions very close to the optimal fronts, while NSGA-II provides only one non-dominated solution. The ZDT3 problem is interesting because it has five discontinuous regions in the Pareto-optimal front. Figure 6.3-c shows all non-dominated solutions obtained after $5 \cdot 10^5$ function evaluations with NSGA-II and SVD-NSGA-II when $n = 50$. From a distribution point of view, there is no difference between the two algorithms: solutions are well distributed for both algorithms. However, the solutions of SVD-NSGA-II are closer to the Pareto optimal front than the ones obtained by NSGA-II. The problem ZDT4 (see Figure 6.3-d) has an exponential number of different local Pareto-optimal fronts in the search space, and only one among them corresponds to the real Pareto front. On this problem, both NSGA-II and SVD-NSGA-II get stuck at different local Pareto-optimal sets, but the convergence and the ability to find a diverse set of solutions are definitely better with SVD-NSGA-II. Finally, Figure 6.3-e shows that SVD-NSGA-II—



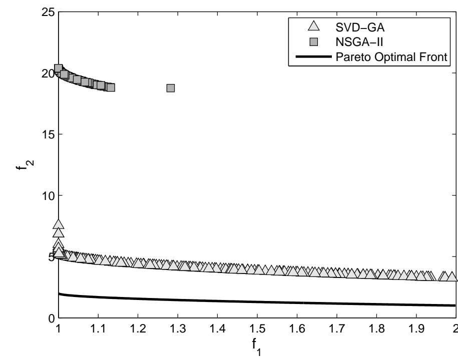
(a) Shifted ZDT1 test problem when $n = 50$



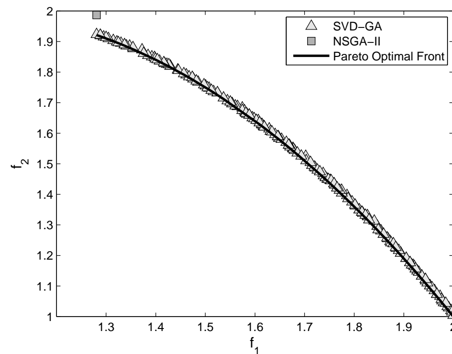
(b) Shifted ZDT2 test problem when $n = 50$



(c) ZDT3 test problem when $n = 50$



(d) Shifted ZDT4 test problem when $n = 50$



(e) Shifted ZDT6 test problem when $n = 50$

Figure 6.3: NSGA-II vs. SVD-NSGA-II on shifted ZDT6 when $n = 50$.

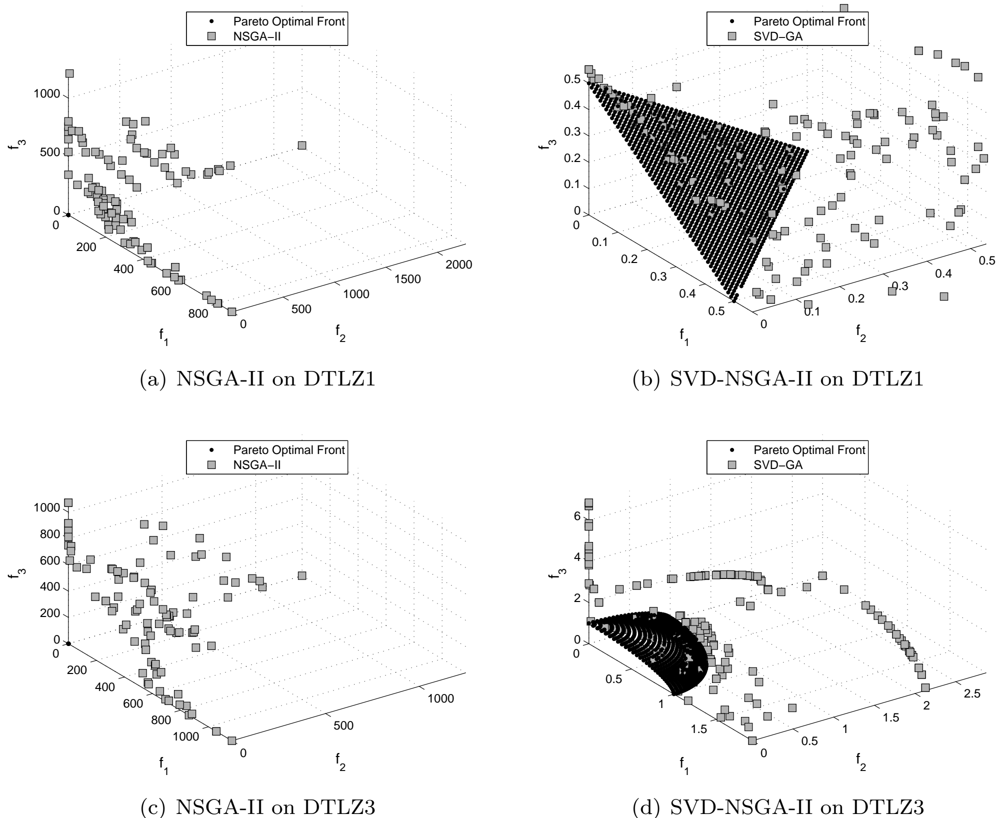
compared to NSGA-II—is able to achieve a better distributed set of non-dominated solutions for ZDT6.

Table 6.7: IGD values achieved by SVD-NSGA-II and NSGA-II on the DTLZ test suite. Values are shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference.

f	n	Func. Eval.	NSGA-II			SVD-NSGA-II		
			Median	Mean	St. Dev.	Median	Mean	St. Dev
DTLZ1	10	$2 \cdot 10^5$	9.52e+2	9.17e+2	2.11e+2	**2.56**	**2.61**	6.24e-1
		$3 \cdot 10^5$	7.14e+2	7.11e+2	1.94e+2	**1.34**	**1.27**	4.19e-1
		$4 \cdot 10^5$	5.36e+2	6.24e+2	2.20e+2	**5.15e-1**	**5.90e-1**	3.26e-1
		$5 \cdot 10^5$	4.96e+2	5.26e+2	1.80e+2	**1.40e-1**	**1.83e-1**	1.73e-1
DTLZ2	50	$2 \cdot 10^5$	6.63e+1	1.08e+2	1.21e+2	**7.25e-2**	**7.36e-2**	3.26e-3
		$3 \cdot 10^5$	4.01	5.82	4.27	**7.12e-2**	**7.22e-2**	3.71e-3
		$4 \cdot 10^5$	5.84e-1	6.50e-1	3.26e-1	**7.05e-2**	**7.10e-2**	2.38e-3
		$5 \cdot 10^5$	1.42e-1	1.71e-1	6.21e-2	**5.45e-2**	**5.79e-2**	7.70e-3
DTLZ3	10	$2 \cdot 10^5$	7.36e+2	7.39e+2	5.96e+1	**5.41**	**5.39**	1.42
		$3 \cdot 10^5$	6.14e+2	6.24e+2	3.95e+1	**2.83**	**3.00**	1.07
		$4 \cdot 10^5$	5.35e+2	5.37e+2	5.37e+1	**1.22**	**1.37**	5.75e-1
		$5 \cdot 10^5$	4.84e+2	4.82e+2	4.74e+1	**5.26e-1**	**6.08e-1**	4.89e-1
DTLZ4	50	$2 \cdot 10^5$	6.99	9.17	7.18	**6.71e-2**	**6.70e-2**	2.05e-3
		$3 \cdot 10^5$	5.74	7.55	6.88	**6.78e-2**	**6.77e-2**	2.24e-3
		$4 \cdot 10^5$	4.13	6.37	6.73	**6.59e-2**	**6.65e-2**	3.29e-3
		$5 \cdot 10^5$	2.77	5.46	6.54	**5.03e-2**	**5.27e-2**	6.05e-3
DTLZ6	50	$2 \cdot 10^5$	1.25e+1	1.22e+1	2.06	**8.54e-2**	**3.82e-1**	3.85e-1
		$3 \cdot 10^5$	8.41	8.27	1.91	**8.40e-2**	**3.82e-1**	3.83e-1
		$4 \cdot 10^5$	6.34	6.46	1.76	**8.13e-2**	**3.81e-1**	3.84e-1
		$5 \cdot 10^5$	5.45	5.51	1.70	**7.00e-2**	**3.56e-1**	3.93e-1

Table 6.7 reports mean, median and standard deviation of the IGD metrics achieved using NSGA-II and SVD-NSGA-II for the three-objectives test problems. The results are also collected at varying function evaluation thresholds. We can observe that, also for the three-objective problems, SVD-NSGA-II outperforms NSGA-II. Indeed, the median and mean IGD values (and the corresponding standard deviation values) achieved by SVD-NSGA-II are always smaller than those obtained by NSGA-II. To provide a visual comparison, we also show examples of simulation results when using NSGA-II and SVD-NSGA-II for some of the three-objective test problems (complete results are reported in Appendix A). All the considered problems have different peculiarities, since they have been designed for testing different abilities of evolutionary algorithms [341], such as convergence speed and distribution of non-dominated solutions.

Figure 6.4 shows the set of non-dominated solutions obtained after $5 \cdot 10^5$ function evaluations using NSGA-II and SVD-NSGA-II for the DTLZ1 problem with $n = 10$. The difficulty in this problem is to converge to the real Pareto front because the search space contains $\left(11^8 - 1\right)$ sub-optimal fronts. The figure shows that NSGA-II is able to maintain solutions that are uniformly distributed in decision search space but very far from the optimal front which is really smaller (it is the black point near the origin of axis). Instead, SVD-NSGA-II is able to obtain uniformly distributed solutions near to the real Pareto front. Similar analysis



(a) NSGA-II on DTLZ1

(b) SVD-NSGA-II on DTLZ1

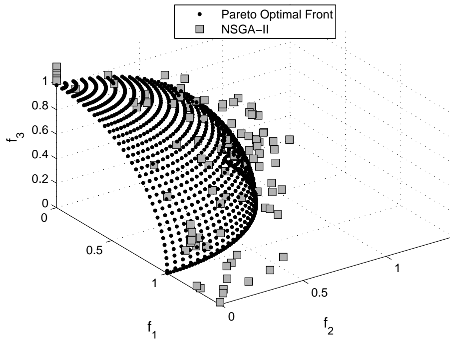
(c) NSGA-II on DTLZ3

(d) SVD-NSGA-II on DTLZ3

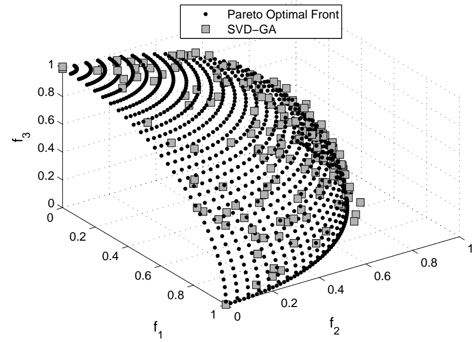
Figure 6.4: NSGA-II vs. SVD-NSGA-II on DTLZ1 and DTLZ2 when $n = 10$.

can be performed for all the other three-objective problems as shown by the results reported in Figures 6.4, and 6.5. Particularly interesting is the DTLZ6 test problem because it has four disconnected Pareto-optimal regions in the search space and it was designed for testing the ability of MOGAs to maintain sub-populations in different Pareto-optimal regions. For such a problem SVD-NSGA-II reached a set of solutions close to the Pareto optimal front and such a solution are well distributed in all the four disconnected Pareto-optimal regions. Instead, NSGA-II provided solutions that are far from the optimal front and cover only two out of four disconnected Pareto-optimal regions.

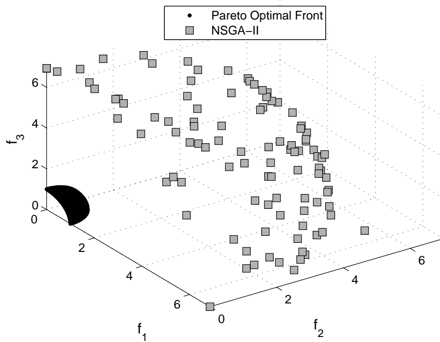
In summary, *SVD-NSGA-II is able to achieve significantly improved convergence speed and distribution of the solution with respect to NSGA-II for both two-objectives and three-objectives test problems.*



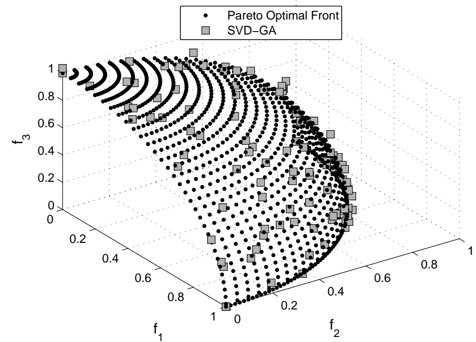
(a) NSGA-II on DTLZ2



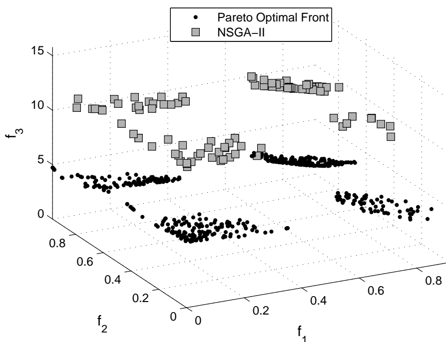
(b) SVD-NSGA-II on DTLZ2



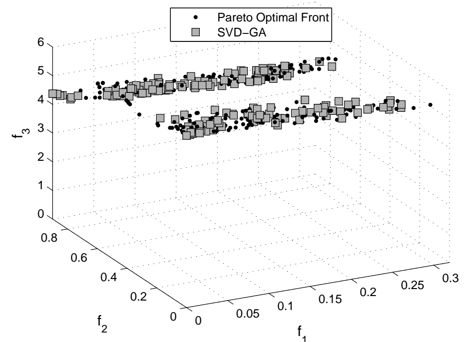
(c) NSGA-II on DTLZ4



(d) SVD-NSGA-II on DTLZ4



(e) NSGA-II on DTLZ6



(f) SVD-NSGA-II on DTLZ6

Figure 6.5: NSGA-II vs. SVD-NSGA-II on DTLZ3, DTLZ4 and DTLZ6 when $n = 50$.

6.5 Conclusion

This Chapter presents SVD-GA and SVD-NSGA-II, i.e. a single-objective GA and NSGA-II extended with a novel diversity-preserving technique based on Singular Value Decomposi-

tion (SVD). In SVD-GA and SVD-NSGA-II, SVD is used to estimate the current *evolution directions* of a population across different generations and to promote the exploration of unexplored regions by creating new individuals with orthogonal evolution directions.

The effectiveness of SVD-GA has been evaluated by solving 15 single-objective benchmark test problems with varying problem dimensions, while the effectiveness of SVD-NSGA-II has been evaluated by solving 12 multi-objective benchmark test problems with varying problem dimensions. The results achieved on two empirical studies indicate the superiority of the proposed algorithms if compared to their original versions.

After the preliminarily evaluation performed in this Chapter, the usefulness of SVD-GA will be further investigate in the context of evolutionary test data generation in Chapter 7. While SVD-NSGA-II will be furthermore investigated in the context of multi-objective test suite optimization in Chapter 8.

Chapter 7

Test Data Generation

Contents

7.1 Introduction . . . 164
7.2 Diversifying the Exploration . . . 165
7.2.1 Basic SVD-GA . . . 167
7.2.2 History Aware SVD-GA . . . 167
7.2.3 Reactive GA . . . 168
7.2.4 Reactive SVD-GA . . . 168
7.3 Implementation . . . 168
7.4 Empirical Evaluation . . . 172
7.4.1 Subjects . . . 172
7.4.2 Research Questions . . . 173
7.4.3 Metrics and Data Analysis . . . 174
7.4.4 GA Parameter Settings . . . 175
7.5 Analysis of the Results . . . 176
7.5.1 $\mathbf{RQ}_1$: *Does orthogonal and/or reactive exploration improve the effectiveness of evolutionary test case generation?* . . . 176
7.5.2 $\mathbf{RQ}_2$: *Does orthogonal and/or reactive exploration improve the efficiency of evolutionary test case generation?* . . . 179
7.6 Threats to validity . . . 181
7.7 Conclusion and future work . . . 183

7.1 Introduction

The application of search-based techniques for automated test case generation has received considerable attention in recent years from the testing community. As a result, a large body of research works has been reported where different meta-heuristic techniques have been applied to address the problem of test data generation [45]. Evolutionary techniques in particular, and specifically GA, have been intensively investigated for the purpose of structural testing [47, 142, 28, 21]. In evolutionary test case generation [21], candidate test cases are encoded into a population of individuals (solutions). These individuals are then evaluated by executing them against the System Under Test (SUT) and their *fitness* (goodness) is measured with respect to a given criterion, e.g., branch coverage. Evolutionary operators such as selection, crossover and mutation are then used to evolve a new *generation* of individuals with *better qualities* than their parents. The process is repeated for several generations either until the criterion is satisfied or the fixed search budget is finished.

Various aspects of evolutionary test data generation, such as selection methods, crossover and mutation operators, fitness evaluation schemes, etc., have been studied in the literature [45], while the problem of genetic drift has been discussed as an issue related with GAs in general [7, 91]. Previous work treated the problems of diversity considering simple heuristics promoting diversity between individuals within the same generation [92, 93]. Hence, new offsprings might explore regions that have been explored in previous generations while new potential regions can still remain unexplored, increasing the likelihood to reach sub-optimal solutions. To address this issue this Chapter investigates the usage of further diversity preserving mechanism to prevent the problem of loss of diversity. Specifically, the goal of this Chapter is to evaluate the SVD-GA algorithm proposed in Chapter 6 in the context of evolutionary test data generation. Such an algorithm —through the estimation of the evolution directions via Singular Value Decomposition (SVD)— is able to augment the population diversity, by periodically introducing new individuals with orthogonal (unexplored) evolution directions.

This chapter gives the following contributions:

1. For the first time, the SVD-based GA is applied to the problem of test case generation, addressing the problem of stagnation during the search process for test input data. To the best of our knowledge, it is the first attempt to explicitly address the genetic drift problem in evolutionary testing.

2. It introduces three variants of SVD-based GA in order to further improve the process of injecting diversity over the basic SVD-GA proposed in Chapter 6.

3. It describes the implementation of these three variants of SVD-GA. The implementation is actually available as an extension of the search based testing tool *EvoSuite*.

4. The evaluation is performed on 17 non-trivial publicly available programs extracted from well-known libraries. The achieved results show that with the appropriate application of diversification techniques, the effectiveness and efficiency of GAs in structural

Nodes	Instructions
s	`example2(int a, int b)`
	`{`
1	`if (a == b)`
	`{`
2	`if (b == 2)`
	`{`
3	`// target statement`
	`}`
	`}`
	`}`

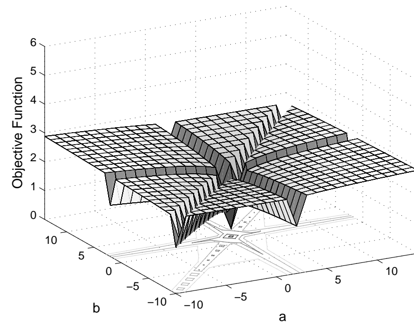


Figure 7.1: Objective function landscape by Wegener *et al.* [21] for the program *example*1.

test data generation can be greatly improved. In particular, effectiveness (coverage) was significantly improved in 47% of the subjects and efficiency (search budget consumed) was improved in 85% of the subjects on which effectiveness remains the same.

Sections 7.2 and 7.3 present the proposed diversification techniques and their implementation, respectively. Sections 7.4 and 7.5 present the design and the results achieved in our empirical study, while the discussion of the threats that could affect the results achieved is reported in Section 7.6. Finally, Section 7.7 points out future works and concludes the Chapter.

7.2 Diversifying the Exploration

The problem of loss of diversity is particularly critical for hard-to-cover branches, for which there are only few values, which allow to traverse the hard branches within the space of all possible input value (large search space but relative small optimal space). For example, let us to consider the simple program shown in Figure 7.1-a. The goal is to execute the true branch of the branching node 2, whose branch predicate is `(b == 2)`. Unless both `a` and `b` are equal to `2` the objective function computed using the combination of *approach level* end *branch distance* is greater than zero. As we can see from the corresponding objective landscape shown in Figure 7.1-b, the objective function contains many local optima and it does not provide any guidance to GAs toward the global optimum (`a=b=2`) because the fitness landscape around it is flat.

Aside from problems of local optima and plateaux appearing in the objective function landscape, the branch distance in some case might guide the search far from the search region containing the global optimum. Let us to consider for example the program shown in Figure 7.2-a. and suppose we want to test the node 5 (i.e., the true branch of the branching node 4), whose branch predicate is `b == 0`. Unless `a` is equal to zero, the objective function computed using the combination of *approach level* and *branch distance* is greater than zero.

Nodes	Instructions
s	`example2(int a)`
	`{`
1	`if (a == 0)`
2	`b=0;`
	`else`
3	`b=1/a`
4	`if (b==0)`
5	`// target statement`
	`}`

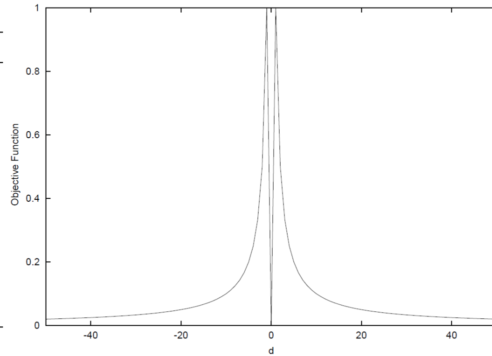


Figure 7.2: Objective function landscape by Wegener *et al.* [21] for the program *example*2.

Moreover, the objective function guides the search away from `b=0` because increasing values of `a` will decrease values of `b` but also will guide the search far from the global optimum that is reached when `a=0`. A further problem can occur in evolutionary test data generation with nested branch conditions, because satisfying a specific branch condition may lead to violate one of the previous branch condition in the same path. In this cases, once input data is found for one or more of the conditions, the chances of finding input data that also satisfy subsequent conditions decreases [67]. This might lead to poor search performance of GAs.

In all these cases, maintaining an adequate level of diversity in the search is a key factor for the effectiveness of a GA. Indeed, a scarcely diversified population can make the genetic operators (crossover and mutation) unable to produce offspring outperforming their parents, with a tendency of guiding the search toward some local optimum. Taking into account the randomness of search space exploration introduced by the mutation and crossover operators, in Chapter 6 we propose the usage of SVD to estimate the directions (*evolution directions*) along which the best individuals in the population are evolving across different generations. Once estimated, these directions are used for promoting the exploration of new regions in the search space, by introducing new individuals with orthogonal evolution directions. In Chapter 6 we also demonstrate the superiority of the resulting SVD-GA over standard GA with distance crowding and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [326, 327] which is one of the fastest algorithm for numerical problems.

In this chapter, we suggest to use the same algorithm, the SVD-GA, in the context of evolutionary test data generation. In particular, starting from its basic definition, we extend the algorithm proposing three variants, in order to make the SVD-based GA more suitable for the test data generation problem:

- *History aware* SVD-GA;
- *Reactive exploration*
- *Combination* of the two variants above.

These extensions are motivated by the fact that the basic SVD-GA relies on the existence of one or more "directions" of evolution across two different generations. Usually in test data generation there are only few input vectors, which cause the coverage of hard-to-cover branches and the fitness landscape around them is often flat. Thus, the associated lack of variation in fitness values leaves the search with no evolution directions (or zero directions), resulting in a high likelihood of failure to find orthogonal test data using the SVD scheme alone. From a preliminary exploratory investigation of SVD-based GA for structural testing, we have noticed that this case is quite common. Hence, to effectively utilize the potential of SVD-based GA in the context of testing, we need to amend this problem. Next subsections describe the three extensions proposed in this chapter to overcome the problem of null evolution direction.

7.2.1 Basic SVD-GA

In the basic formulation of SVD-GA, the evolution directions are estimated by comparing the SVD decompositions of a population P before and after k generations. Formally, let P_t and P_{t+k} be two different populations generated by GA at generations t and $t+k$, respectively (where k is a user defined parameter). SVD is computed on both populations, resulting in matrices that represent evolution directions and magnitudes: $P_t = (U_t \cdot \Sigma_t \cdot V_t)$ and $P_{t+k} = (U_{t+k} \cdot \Sigma_{t+k} \cdot V_{t+k})$ where $V \in \mathbb{R}^{n \times n}$ and $\Sigma \in \mathbb{R}^{m \times n}$. The column vectors of V represent the main directions in which the population is distributed while the diagonal matrix Σ represents the importance of each direction in V. Computing $\overline{V} = V_{t+k} - V_t$ gives a precise indication of the directions of evolution of the population, while $\overline{\Sigma} = \Sigma_{t+k} - \Sigma_t$ indicates their magnitude. New individuals in orthogonal spaces are generated as follows[1]:

$$P^*_{t+k} = U_{t+k} \cdot (\Sigma_{t+k} + \overline{\Sigma}) \cdot \left(V_{t+k} + (\overline{V}^o)\right)^T \tag{7.1}$$

Once the new orthogonal individuals are generated, the worst individuals in the last generation P_{t+k} are replaced by these news individuals, thus, introducing diversification through new individuals having orthogonal and potentially unexplored directions.

7.2.2 History Aware SVD-GA

Orthogonal exploration of the search space relies on the fact that there is a "direction" of evolution in the search between the individuals in the populations at times t and $t+k$. The first extension of SVD-based GA is represented by a *history aware* approach, in which we consider not only the population at generation t and $t+k$, but also the *history* of older populations encountered during evolution. Specifically, if $\overline{V}$, computed between P_{t+k} and P_t, gives a zero matrix (i.e. the there is no evolution direction), we compute $\overline{V}$ between population P_{t+k} and population $P_{t-k}, P_{t-2k}, P_{t-3k}, \ldots$, until we achieve a non-zero direction matrix or we consume the available history entirely. In other words, we propose to compute

[1] For further details, see Chapter 6.

the evolution directions starting from the last generation $t+k$ going back across the previous generations until we reach a non-null evolution direction (or we exhaust the history). For practical purposes, we consider a bounded history of snapshots: $P_t, P_{t-k}, P_{t-2k}, \ldots, P_{t-hk}$, with $h \geq 0$ a small, user defined constant. In our empirical experiments, the value $h = 10$ gives reasonably good results. In the following, this approach is referred to as *H-OV-GA*, i.e. *History aware* SVD-GA.

7.2.3 Reactive GA

Another way to ensure diversity in the exploration of the search space consists of taking a random direction, rather than an orthogonal direction, when null evolution direction. In this variant, we compute the evolution direction $\overline{V}$ by comparing the SVD decomposition between P_{t+k} and P_t, similarly as done for the basic SVD-GA. However, differently from H-OV-GA, if the current evolution direction is null —which happens when $\overline{V} = 0$ holds— we consider such a scenario as an indicator of stagnation and a *random walk* is taken from the actual state by generating a random direction matrix, which is used instead of $\overline{V}^o$ in equation 7.1. More precisely, if $\overline{V} = 0$ then new individuals are generated by modifying Equation 7.1 as follows:

$$P^*_{t+k} = U_{t+k} \cdot (\Sigma_{t+k} + \overline{\Sigma}) \cdot (V_{t+k} + Rand)^T \tag{7.2}$$

where $Rand \in \mathbb{R}^{n \times n}$ is a randomly generated matrix representing a new random direction. Otherwise, the search continues normally without any random or reactive exploration. In the following, this approach is referred to as *R-GA*, i.e *Reactive* GA because when the stagnation is identified through SVD, GA reacts by taking a random walk in the search space.

7.2.4 Reactive SVD-GA

While the two approaches discussed in the previous sub-sections are expected to improve over the basic algorithm in diversifying the search, their combined effect could be even better than either of them applied separately. Thus, we propose yet another SVD-based GA variant which combines the *history aware* approach with the *reactive exploration.* Specifically, we first compute the direction matrix $\overline{V}$ between the current populations P_{t+k} and P_t, as previously discussed. If this direction matrix is non-zero, we compute an orthogonal matrix $\overline{V}^o$ and generate a new population by applying Equation 7.1. Otherwise, if the direction matrix $\overline{V}$ is equal to zero —i.e. there is no evolution direction— we generate a random direction matrix and generate a new population by applying Equation 7.2. In the following, this approach is referred to as *R-OV-GA*.

7.3 Implementation

While the proposed orthogonal exploration approach can be applied to any numeric program, whether it is object oriented or not, for the sake of demonstration we present the implemen-

tation of the technique in the context of object-oriented programs. Specifically, we assume the goal is to achieve full *branch coverage* of a Class Under Test (CUT).

In this chapter we consider object-oriented programs with either numeric primitive input data (i.e., as `int`, `double` and `long` variables), boolean variables and arrays/matrices of primitive types (i.e., arrays of `int`, `double`, `long` and `boolean`). We plan to extend our approach to variable size input vectors and to non-numeric data as part of our future work. To simplify the application of SVD-GA and its variants to object-oriented programs, we make the assumption that any method invocation sequence necessary to prepare the class under test for the execution of the method under test is performed inside the class constructor. In other words, we suppose that the constructors of the class under test instantiate the required objects and attribute before executing a method under test that requires them as precondition. If no such constructor exists, we define one constructor just for the purposes of testing.

Solution encoding. Let us to consider a generic class `Class` under test having a constructor `Class(...)` with i input parameters, and a method `method(...)` to be tested with j input parameters. A candidate solution (test case) is encoded in two parts: a *genotype* used to store the input data and a *phenotype* that store the statements of the corresponding test case. Specifically, the *genotype* is a numerical vector of size $n = i + j$ which represents the set of numeric values that have to be generated by a GA. The *phenotype* is a sequence of Java statements (instantiated with the input values stated in the genotype) that are actually executed against the class under test via JUnit. Consequently, all test cases for the method `method` are represented as vector of the same size $n = i + j$, while a population of m test cases (individuals) can be represented as an $m \times n$ numerical matrix that lends itself to be used for computing the SVD decomposition (for H-OV-GA, R-GA and OV-GA).

For a generic class `C` having more constructors $(C_1, \ldots, C_s)$ and more methods (m_1, ..., m_r), we first randomly select one constructor (e.g. C_1) and one method (e.g. m_1) from the pool. Thus, we statically generate the corresponding phenotype with two statements: a call to the selected constructor C_1 and a call to selected method m_1. Once the phenotype is built, we construct the corresponding genotype as an input vector which store both the parameters for C_1 and parameter for m_1. Thus, we run GA (or H-OV-GA, R-GA and R-OV-GA) in order to find the set of genotypes, which allows to cover all the branches of the method under test m_1. The test cases are then written instantiating the (unique) phenotype with the parameters stored in the genotypes obtained by GA. The process is then repeated for each pair of methods and constructors in `C` until all methods and all constructor have been tested.

When the method under test `m` is not a publicly visible method (e.g., it is a private method), we select a public method `m'` which (directly or indirectly) calls `m`, if one exists. Otherwise, the target is deemed unreachable. If `m` is a static method, then the individual will be composed of only the j parameters of `m`. The actual test case will then be composed of statements for instantiating the class, followed by an invocation of method `m` on the instance, or simply the static invocation of `m`, if it is a static method.

```
public class A {
 private int a, b, c;

 public A (int a, int b, int c) {
   this.a = a;
   this.b = b;
   this.c = c;
 }

 public void m1(int x){
    ...
    }

 public static int m2 (double d1, double d2, double[] d3){
    ...
    }
}
```

Figure 7.3: An example of class under test

For example consider the class `A` in Figure 7.3 and a target method `m1`. A candidate solution could have the genotype equal to `(3, 4, 5, 0)` while the corresponding phenotype test case is:

```
A A0 = new A (3, 4, 5);
A0.m1 (0);
```

For a target method `m2`, a candidate solution could be `(10.5, 12.0, 12.0, 0.5, 1.25)` (arrays are instantiated with a fixed, predefined length) with the corresponding test case:

```
double[] double0 = new double[3];
double0[0] = 12.0;
double0[1] = 0.5;
double0[2] = 1.25;
int int0 = A.m2 (10.5, 12.0, double0);
```

Objective function. Fitness evaluation is based on the combination of approach level and branch distance metrics [45] in which the test case associated with an individual is executed against the instrumented CUT. *Approach level* measures how many control branches away the current individual is from hitting the target branch, while *branch distance* measures how far it is from making the branch condition true, where the considered branch is the first one which makes the target no longer reachable. After the execution, the fitness of each individual is computed as the sum of the approach level and the normalized branch distance.

Algorithm 10: Main algorithm

```
1 begin
2     while total_budget not finished do
3         for every target not covered do
4             budget ← total_target/uncovered_targets;
5             solution ← generateSolution(target, budget);
6             suite ← suite ∪ solution;
7     return suite
```

This value is minimized, with value zero associated with the target branch being covered by the individual. A detailed description this fitness function can be found in Section 5.2.3.

Search process. The search process is described in Algorithm 10 which distributes the available test case generation budget (measured in number of statements executed) across the targets yet to be covered and relies upon *generateSolution* to obtain an input vector to cover the selected target. The routine *generateSolution* corresponds to the specific genetic algorithm run in order to find the test case to cover the target *target* using a specific search budget *budget*. For the proposed variants to SVD-GA, *generateSolution* includes the common steps performed in evolutionary testing (generation of an initial random population and its evolution by selection, mutation and crossover, depending on the fitness values of the individuals). In addition to such steps, the algorithm periodically (after k generations, with k a user defined parameter) performs orthogonal diversification of the population. Based on a history of length h, an orthogonal population P_t^* is produced and is used to replace the individuals with lowest fitness values. New individuals are generated according to the specific instance of the *generateSolution* algorithm, which can be either H-OV-GA, R-GA or R-OV-GA.

Genetic operators. Genetic operations are applied to the vector representation of an individual. In particular, crossover between two individuals is performed by exchanging portions of the vectors representing the genotype. Mutation on an individual changes one or more values from the vector. Each value in the vector is subject to mutation with probability $1/len$ where len is the length of the vector representing the genotype. The actual mutation operation performed on a value selected for mutation could be one of the following three operations depending on their respective probabilities:

- negate: it changes the sign of a double, long or int variable; it also changes the value of boolean variable (similar to bit-flip mutation);
- increment: a randomly generated delta value is added/subtracted to/from the value;
- replace from pool: replace by a randomly selected value from a pool of previously used values.

Elitism is used, so as to ensure that the elite of each generation is preserved.

Table 7.1: Subjects used in our study.

No.	Name	Origin	Methods	Coverage Goals
1	ArithmeticUtils	Apache Commons Math	10	99
2	Arrays	Java Collections	8	75
3	Beta	Apache Commons Math	11	90
4	CreditCardValidator	Apache Commons Valid.	4	32
5	Complex	Apache Commons Math	29	126
6	FastMath	Apache Commons Math	3	60
7	Fraction	Apache Commons Math	28	108
8	IPAddressValidator	Apache Commons Valid.	3	243
9	LUDecomposition	JAMA library	8	76
10	KolmogorovSmirnov Distribution	Apache Commons Math	6	50
11	QRDecomposition	JAMA library	6	72
12	Quadratic	[46, 342]	1	7
13	RootsOfUnity	Apache Commons Math	6	27
14	SaddlePointExpansion	Apache Commons Math	3	16
15	Sort	SIR	15	70
16	Tomorrow	[46, 342]	1	107
17	TriangularDistribution	Apache Commons Math	13	50

Tool. Our prototype tool for the branch coverage testing of Java classes is implemented as an extension of the EvoSuite [287] tool and is available for download from www.evosuite .org/orthogonal-ga.

7.4 Empirical Evaluation

This section describes the study we conducted aiming at evaluating whether the performance of evolutionary test case generation improves when the employed GA is enriched with orthogonal and/or reactive exploration mechanisms. The quality focus of the study is the effectiveness and the efficiency of the test case generation process based on the proposed approach compared to standard GA (St-GA hereafter), while the perspective is of researchers aiming at developing more effective automatic testing tools, that could be used by software engineers for reducing the time/cost required for testing a software system.

7.4.1 Subjects

The experimental study is performed on 17 publicly available Java classes with numerical input data type. Table 8.2 shows the characteristics of the classes considered in our study in terms of number of methods and number of coverage goals (i.e., branches).

Most of the classes (9) were extracted from the open source *Apache.commons.math* library, which contains several mathematical and statistical Java classes addressing the most common problems not available in the Java standard libraries or in Commons Lang[2]. We also considered 2 classes extracted from the *Apache.commons.validator* library, which contains classes for checking the validity of input data. The class *Sort* was extracted from the Software artifact Infrastructure Repository (SIR) [343] and it implements different sorting algorithms, such as insertion sort and quick sort. *Quadratic* and *Tomorrow* are two classes widely used in previous studies on evolutionary test case generation [46, 342]. Finally, *LUDecomposition* and *QRDecomposition* were extracted from the JAva MAtrix (JAMA) library, which provides a suite of classes for performing linear algebra operations[3], while *Arrays* was extracted from the well-known Java Collections framework.

Whenever necessary, minor manual adjustments were performed on the classes for the purpose of the study, making sure that the classes are not significantly changed. Most of the modifications are superficial and they concern methods with non-numeric parameters, which are currently not supported by our prototype tool. In such cases, either a public wrapper method is added in which we pass a fixed value for the non-numeric parameters, or we simply drop the method if it is trivial (for e.g. a setter method).

7.4.2 Research Questions

In the context of our study, we formulated the following research questions:

- **RQ**$_1$: *Does orthogonal and/or reactive exploration improve the* ***effectiveness*** *of evolutionary test case generation?* This is the main research question of our study. Specifically, we aim at evaluating to what extent the proposed orthogonal and/or reactive exploration mechanisms improve the effectiveness (coverage) of GA-based test case generation.

- **RQ**$_2$: *Does orthogonal and/or reactive exploration improve the* ***efficiency*** *of evolutionary test case generation?* With this second research question, we aim to analyze to what extent the proposed approaches are able to reduce the cost required for reaching the highest coverage.

Note that we analyze the efficiency of the proposed approach only when we do not observe a significant improvement in terms of effectiveness. This is because during evolutionary test case generation a budget (maximum cost) is fixed *a priori*. Thus, every test suite generated within such a budget is acceptable from the point of view of the efficiency. This means that efficiency plays a secondary role compared to the effectiveness. In other words, a solution with a better effectiveness and a higher cost (within the available budget) is preferred to another solution with a reduced cost but worse effectiveness. On the other hand, when the

[2]http://commons.apache.org/math/
[3]http://math.nist.gov/javanumerics/jama/

effectiveness (coverage) is the same, the technique with highest efficiency (lower execution time) is preferable.

In order to answer our research questions, we compared test case generation based on St-GA with the following variants (defined in Section 7.2):

- *GA with history aware orthogonal exploration* (H-OV-GA): St-GA enriched with the orthogonal exploration mechanism based on SVD and spanning the history of populations generated over time.
- *GA with reactive exploration* (R-GA): St-GA enriched with the reactive exploration mechanism based on stagnation analysis (i.e., lack of evolution direction).
- *GA with both orthogonal and reactive exploration* (R-OV-GA): St-GA enriched with the combination of the previous two variants, H-OV-GA and R-GA.

7.4.3 Metrics and Data Analysis

The effectiveness of the different approaches ($\mathbf{RQ}_1$) was measured in terms of branch coverage:

$$Coverage = \frac{\text{\#covered branches}}{\text{\#total branches to be covered}}$$

while the efficiency ($\mathbf{RQ}_2$) in terms of consumed search budget (number of statements executed) during the search for test data. The higher the coverage, the better the effectiveness of the test data generation technique. Vice versa, the lower the number of executed statements required to reach the best coverage, the higher the efficiency of a technique.

All the approaches have been executed 100 times on each class to account for the inherent randomness in GA. Then we collected the branch coverage and the number of statements executed to reach the best coverage for each run, within a given maximum budget limit of time/cost (see Section 7.4.4).

In order to provide statistical support for the results, we performed statistical tests to check whether the effectiveness (or efficiency) achieved by one of the novel approaches (e.g., R-OV-GA) is significantly better than that reached by St-GA. Specifically, we used the Wilcoxon Rank Sum test [226] to test the following null hypotheses:

- H_{0_1}: *there is no difference between the effectiveness of H-OV-GA/R-GA/R-OV-GA and St-GA.*
- H_{0_2}: *there is no difference between the efficiency of H-OV-GA/R-GA/R-OV-GA and St-GA.*

For H_{0_1} (effectiveness), the dependent variable is represented by the number of branches covered by an evolutionary test data generation algorithm over the 100 independent runs. Instead, for H_{0_2} (efficiency) the dependent variable is represented by the number of executed statements. Results are intended as statistically significant at $\alpha = 0.05$. Since this requires

performing three tests for each subjects, we adjusted the p-values using Holm's correction procedure [227].

Besides testing the null hypotheses, it is of practical interest to estimate the magnitude of the difference, in terms of branch coverage or consumed budget, achieved by the different methods. For this purpose, we use the Cohen d effect size [344], which measures the magnitude of the effect of the different algorithms on the dependent variables. It is defined as the difference between the means, divided by the standard deviation of the differences between the coverage (number of statements) values achieved by two GA variants (e.g., St-GA and R-OV-GA)

$$d = \frac{mean(Coverage_{GA_1}) - mean(Coverage_{GA_2})}{stdev(Coverage_{GA_1} - Coverage_{GA_2})}$$

The effect size is considered large for $d \geq 0.8$, medium for $0.5 \leq d < 0.8$ and small for $0.2 \leq d < 0.5$ [344]. This provides a classification (large, medium, small) of the magnitude of the difference, thus, it is quite easy to be interpreted.

7.4.4 GA Parameter Settings

To set the parameters of the St-GA algorithm and of the variants investigated in this study, we conducted a preliminary set of runs, aimed at assessing the sensitivity of subjects and techniques to the parameters. The same configuration was used for all the experimented variants (St-GA, H-OV-GA, R-GA and R-OV-GA), since no major sensitivity to the parameters was observed.

With regards to subjects, for most of the parameters a consistently similar set of values was used for all subjects, since the observed sensitivity was minor. However, for two parameters (Population size and SVD-frequency), some major differences were observed. Hence, we group the subjects into two and use a different value for each group. The specific values of the important parameters we used in our experiments are shown below:

- **Population size**: we choose moderate population sizes of 20 for one group of subjects and 40 for the other.
- **Initial population**: for each subjects the initial population is uniformly and randomly generated within the search space $[-2048, 2048]^N$.
- **Search budget**: we restrict the search budget to a maximum number of 10E+6 executed statements or a maximum of 30 minutes of computation time for fitness evaluations. For subjects `LUDecomposition` and `QRDecomposition` however, because of their complexity, we used a higher budget limit of 10E+7.
- **Crossover**: we use single point fixed crossover with probability $P_c = 0.75$.
- **Mutation**: we use a uniform mutation function with probability $P_m = 1/len$ where len is the dimension of the vector representing the individual.

- **Selection function**: rank selection is used with $bias = 1.7$.
- **Elitism**: the number of individuals in the current generation that are kept alive across the next generation is 1.
- **SVD-frequency**: orthogonal sub-populations are generated by SVD every k generations where $k = 1$ for one group of subjects and $k = 2$ for the other. This parameter does not apply to St-GA.
- **SVD-proportion**: the proportion of individuals in the current population to be considered for estimating the evolution directions is set to $P_{SVD} = 0.25$.
- **Historic Data**: the number of generations to be considered for detecting evolutionary stagnation is $h = 10$.

The settings have been calibrated using a trial-and-error procedure, and some of them (i.e., crossover and mutation probabilities) are values commonly reported in the literature.

7.5 Analysis of the Results

This section discusses the results of our study in light of the research questions formulated in the previous section.

7.5.1 RQ$_1$: *Does orthogonal and/or reactive exploration improve the effectiveness of evolutionary test case generation?*

Table 7.2 reports the mean branch coverage values (over 100 independent runs) achieved by St-GA, H-OV-GA, R-GA and R-OV-GA for all the subjects. The table reports both (i) the mean number of branches and (ii) the mean percentage of the total branches covered by each algorithm.

As we can see, in 8 out of 17 cases the mean coverage value achieved by R-OV-GA is higher then that achieved by St-GA (with an improvement ranging between 0.20% and 20%). Only on `SaddlePointExpansion` and `TriangularDistribution` the effectiveness of all the experimented approaches is comparable and no clear winner can be identified. It is worth noting that on the other classes (5 out of 17) there is no difference between the experimented methods, because all the approaches achieved 100% of branch coverage.

From the analysis of the results, it also emerges that the mechanism that combines the orthogonal exploration and the reactive exploration (R-OV-GA) often turned out to be more effective than either of the two mechanisms taken individually, i.e., H-OV-GA and R-GA. Indeed, in only 4 cases (i.e., when testing `Arrays`, `SaddlePointExpansion`, `TriangularDistribution` and `KolmogorovSmirnovDistribution`) R-GA and H-OV-GA achieved a (slightly) better coverage than R-OV-GA. However, in two of these cases all the algorithms achieved almost the same level of coverage and none of the experimented algorithms is able to achieve 100% coverage.

Table 7.2: Average coverage values achieved by St-GA, R-OV-GA, H-OV-GA and R-GA over 100 independent runs. Values in bold face turned out to be the best against those achieved by the other algorithms for the same CUT.

Class under test	St-GA		H-OV-GA		R-GA		R-OV-GA	
	Cov	Cov %	Cov	Cov %	Cov	Cov %	Cov	Cov %
ArithmeticUtils	85.81	86.68%	85.89	86.76%	85.77	86.64%	**85.93**	**86.80%**
Arrays	60.20	80.27%	**62.46**	**83.28%**	59.74	79.65%	62.02	82.69%
Beta	76.91	85.46%	76.93	85.48%	76.98	85.53%	**77.00**	**85.56%**
Complex	73.39	58.25%	78.59	62.37%	78.56	62.35%	**78.77**	**62.52%**
CreditCardValidator	30.39	94.97%	31.41	98.16%	30.43	94.63%	**31.69**	**98.90%**
FastMath	36.83	61.38%	38.82	64.70%	37.48	62.47%	**40.20**	**67.00%**
Fraction	88.37	81.82%	88.41	81.86%	88.39	81.84%	**88.48**	**81.93%**
KolmogorovDistribution	37.93	75.86%	**39.07**	**78.13%**	37.70	75.40%	37.06	74.13%
IPAddressValidator	243.00	100.00%	243.00	100.00%	243.00	100.00%	243.00	100.00%
LUDecomposition	53.55	70.46%	57.76	76.12%	56.12	73.84%	**60.93**	**80.18%**
QRDecomposition	53.04	73.67%	61.94	86.03%	56.83	78.93%	**66.64**	**92.56%**
Quadratic	7.00	100.00%	7.00	100.00%	7.00	100.00%	7.00	100.00%
RootsOfUnity	27.00	100.00%	27.00	100.00%	27.00	100.00%	27.00	100.00%
SaddlePointExpansion	14.00	87.50%	14.00	87.50%	**14.01**	**87.56%**	14.00	87.50%
Sort	70.00	100.00%	70.00	100.00%	70.00	100.00%	70.00	100.00%
Tomorrow	106.90	99.99%	106.90	99.99%	107.00	100.00%	107.00	100.00%
TriangularDistribution	42.33	84.66%	**42.57**	**85.14%**	42.24	84.48%	42.55	85.10%

It is important to note that the occurrence of no or a marginal improvement in terms of coverage does not necessary indicate that the proposed techniques are not useful. Indeed, by inspecting the code of such subjects, we noted that often it is impossible to achieve a higher coverage, since the uncovered branches are not reachable. For instance, let us consider the class `Beta` where R-OV-GA seems to be able to improve branch coverage just by 0.1% as compared to St-GA. By inspecting the code, we noticed that this is the highest possible coverage, since the uncovered branches cannot be covered by any input. Figure 7.4 shows an example of unreachable branch. The private method `logGammaMinusLogGammaSum` contains an `if` instruction for checking the validity of the first input parameter `a`. This private method is called by the public method `logBeta` which independently performs exactly the same control before calling the private method. Hence, the method `logGammaMinusLogGammaSum` will never throw the `NumberIsTooSmallException` because, on the basis of the method visibility, the public method `logBeta`, which is the only caller, will never pass an invalid parameter to it.

Figure 7.5-a shows a boxplot of the experimented methods for one of the subjects: *QRDecomposition*. It can be seen from the boxplot that the distribution of coverage values obtained by R-OV-GA over all the independent runs is substantially higher then the distribution achieved by all the other methods. Specifically, in more than 95% of independent runs R-OV-GA achieved a coverage value ranging between 92% and 97%. Vice versa, St-GA reached a coverage value lower than 80% in the majority of all the independent runs. H-OV-GA and R-GA turned out to be quite better than St-GA, but significantly worse than R-OV-GA. To provide a further evidence of the benefits introduced by diversity, Figure 7.5-b compares

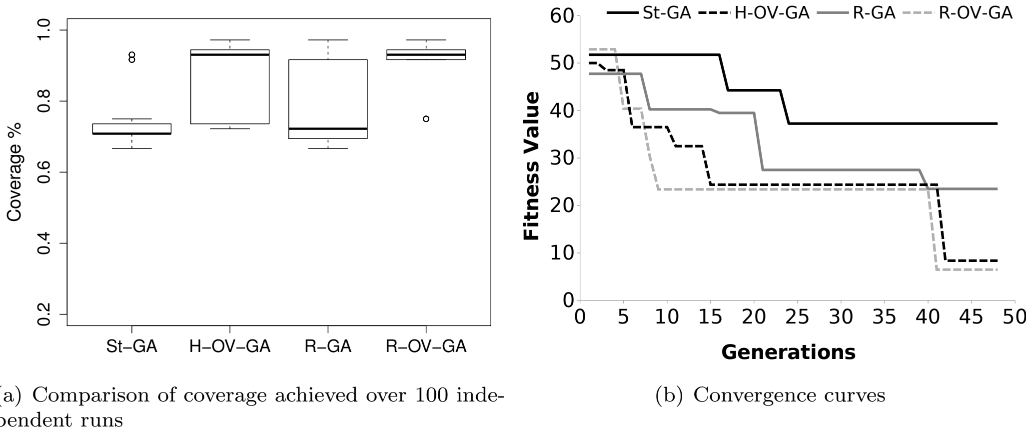
```
private static double logGammaMinusLogGammaSum (double a, double b)
        throws NumberIsTooSmallException {
if (a < 0.0) {
    throw new NumberIsTooSmallException(a, 0.0, true);
    }
...
}

public static double logBeta(double p, double q) {
if (Double.isNaN(p) || Double.isNaN(q) || (p <= 0.0) || (q <= 0.0)) {
                   return Double.NaN;
                   }
...
final double a = FastMath.min(p, q);
final double b = FastMath.max(p, q);
...
return logGammaMinusLogGammaSum(a, b);
...
}
```

Figure 7.4: Example of unreachable branch for *Sort*.

the best fitness values achieved by the four experimented algorithms for *QRDecomposition*. It can be seen how R-OV-GA, R-GA and H-OV-GA converge more quickly toward better solutions, while St-GA has a lower convergence rate and is still trapped in some local optima after 23 generation. Finally, R-OV-GA seems to be able to combine the orthogonal exploration and reactive exploration showing a better ability to find better solutions than its constituents, i.e. H-OV-GA and R-GA.

All these considerations are also supported by the statistical analysis. Table 7.3 reports the results of the Wilcoxon tests (after correcting the p-values using Holm's correction) and the Cohen d effect size. As we can see, in 8 out of 17 cases R-OV-GA provides a statistically significant improvement in terms of effectiveness, while for the remaining cases there is no statistically significant difference between the experimented methods. However, in 5 out of 17 cases it is impossible to improve the coverage since St-GA (as well as the other variants) achieves 100% of coverage. This means that R-OV-GA is able to improve the coverage in 70% of cases where improvement is possible.

As for the magnitude of the improvement, when comparing R-OV-GA with St-GA the effect size is generally large and only in two cases it is medium and small, respectively. It is worth noting that in the cases where there is no statistically significant difference between St-GA and R-OV-GA, the effect size is always positive. This means that R-OV-GA never reached a coverage value lower than St-GA. Similar considerations can be derived from the comparison of H-OV-GA and St-GA. When comparing St-GA and R-GA, Table 7.3 reveals



(a) Comparison of coverage achieved over 100 independent runs

(b) Convergence curves

Figure 7.5: Comparison of St-GA, H-OV-GA, OV-GA and R-GA over on QRDecomposition.

that the reactive exploration is able to significantly improve the effectiveness of St-GA with an effect size that varies over the tested classes, ranging from very large ($d \gg 1$) to small. There are also four cases where the effect size is negative, indicating that the reactive exploration taken individually does not always improve (sometimes marginally worsens) the effectiveness of GA.

In summary, *for* $\boldsymbol{RQ}_1$ *we can assert that the proposed mechanisms for increasing the population diversity (and reducing the stagnation probability) are able to improve the effectiveness of GA-based evolutionary test case generation. In addition, among the different variants of diversification mechanisms, R-OV-GA (that combines orthogonal and reactive exploration mechanisms) turned out to be the best.*

7.5.2 RQ$_2$: *Does orthogonal and/or reactive exploration improve the efficiency of evolutionary test case generation?*

In cases where there was no statistically significant improvement of effectiveness, we analyzed the search budget consumed by each technique and investigated whether there is any improvement in terms of efficiency. Table 7.4 reports the mean number of statements executed by St-GA, R-GA, H-OV-GA and R-OV-GA for reaching the best coverage. As one can see from the table, in the majority of the cases (6 out of 7 cases) R-OV-GA reached the best mean efficiency, with a cost reduction ranging from 20% to 80% on average over 100 independent runs. This means that the combined application of orthogonal and reactive exploration achieves the same coverage as St-GA, but with a cost reduction of about 40% on average (which indicates a substantial improvement in terms of efficiency). The only exception is represented by the class *Fraction*, where the cost is slightly higher.

The analysis of the results also reveals that H-OV-GA and R-GA are not able to make

Table 7.3: P-values achieved using Wilcoxon Rank Sum test with Holm's correction and Cohen d effect sizes. We use S, M, and L to indicate small, medium and large effect sizes respectively.

Class under test	**H-OV-GA¡St-GA**		**R-GA¡St-GA**		**R-OV-GA¡St-GA**	
	p-value	Cohen d	p-value	Cohen d	p-value	Cohen d
ArithmeticUtils	0.31	0.08	0.48	-0.05	0.17	0.13
Arrays	< **0.001**	0.83 (L)	0.87	-0.05	< **0.001**	0.77 (M)
Beta	0.6	0.03	**0.04**	0.09	**0.006**	0.12
Complex	< **0.001**	4.35 (L)	< **0.001**	6.58 (L)	< **0.001**	6.70 (L)
CreditCardValidator	< **0.001**	1.29 (L)	0.30	-0.09	<**0.001**	1.90 (L)
FastMath	< **0.001**	0.78 (L)	**0.04**	0.22 (S)	< **0.001**	1.21 (L)
Fraction	0.89	0.02	0.89	-0.01	0.89	0.07
IPAddressValidator	-	-	-	-	-	-
LUDecomposition	**0.04**	0.75 (M)	**0.01**	0.66 (M)	< **0.001**	0.96 (L)
KolmogorovDist.	0.44	0.06	0.40	0.01	0.49	0.08
QRDecomposition	< **0.001**	1.82 (L)	0.20	0.14	< **0.001**	2.94 (L)
Quadratic	-	-	-	-	-	-
RootsOfUnity	-	-	-	-	-	-
SaddlePointExp.	-	-	-	-	-	-
Sort	-	-	-	-	-	-
Tomorrow	0.48	0.01	-	-	0.48	0.01
TriangularDist.	**0.01**	0.26 (S)	0.24	-0.09	**0.01**	0.26 (S)

statistically significant improvements as compared to St-GA in terms of efficiency. Indeed, for H-OV-GA the amount of consumed budget is reduced in 5 out 7 cases, while it is increased in the remaining 2 cases. Similarly, R-GA showed a reduction in terms of consumed budget in only 4 out of 7 cases. Figure 7.6 shows a boxplot of the consumed search budget by the experimented methods for one of the subjects: the class *Sort*. The boxplot shows that the distribution of cost values obtained by R-OV-GA over all the independent runs is quite lower then the distribution achieved by all the other methods. Specifically, the budget consumed by R-OV-GA is lower than the budget required by the other algorithms for the majority (about the 75%) of the runs.

To provide statistical support for the above considerations, Table 7.5 reports the results of the Wilcoxon tests (after correcting p-values using the Holm's correction) and the corresponding Cohen d effect size. As we can see, from the point of view of efficiency R-OV-GA outperforms St-GA in a statistically significant way in 6 out 7 cases, with an effect size that is large ($d > 0.8$) in the majority of the cases. In the remaining case, the Wilcoxon test reveals that the difference is not statistically significant, even if the corresponding effect size is marginally positive. As expected, for H-OV-GA and R-GA there is a statistically significant difference in only few cases: 4 out of 7 cases and 2 out of 7 cases, respectively.

In summary, *for* $\boldsymbol{RQ_2}$ *we can assert that R-OV-GA is able to obtain the same effectiveness as St-GA but with a cost reduction of about 40% on average. Among the different variants of exploration mechanisms proposed, R-OV-GA provides the best efficiency as com-*

Table 7.4: Average budget consumed by St-GA, R-OV-GA, H-OV-GA and R-GA over 100 independent runs (the percentage variations with respect to St-GA are also shown). Values in bold face turned out to be the best (lowest) against those achieved by the other algorithms for the same CUT.

Subject	St-GA	H-OV-GA		R-GA		R-OV-GA	
	Cost	Cost	Var %	Cost	Var %	Cost	Var %
Fraction	**290606**	301099	+4.36%	336645	+15.84%	308619	+6.20%
IPAddressValidator	330590	223959	-32.25%	343652	+3.95%	**190848**	**-42.27%**
KolmogorovDist.	56340	48003	-14.80%	37138	34.08%	**36539**	**-35.15%**
Quadratic	2116	1555	-26.51%	2260	+6.81%	**1458**	**-31.10%**
RootsOfUnity	7112	6633.3	-6.74%	6087.41	-14.41%	**4531**	**-36.24%**
SaddlePointExpansion	70822	71381	+0.79%	66120	-6.64%	**54725**	**-22.73%**
Sort	581837	574848	-1.20%	569851	-2.06%	**464132**	**-20.23%**
Tomorrow	405251	76388	-81.15%	371080	-8.43%	**76032**	**-81.24%**

Table 7.5: P-values achieved using Wilcoxon Rank Sum test with Holm's correction and Cohen *d* effect sizes. Also, we use S, M, and L to indicate small, medium and large effect sizes respectively.

Class under test	H-OV-GA ¡ St-GA		R-GA ¡ St-GA		R-OV-GA ¡ St-GA	
	p-value	Cohen d	p-value	Cohen d	p-value	Cohen d
Fraction	0.89	0.11	0.89	0.18	0.89	0.04
IPAddressValidator	< **0.001**	0.53 (M)	0.61	-0.05	< **0.001**	0.89 (L)
KolmogorovDist.	0.86	0.03	0.11	0.29	**0.03**	0.40 (S)
Quadratic	< **0.001**	0.55 (M)	< **0.001**	0.42 (S)	< **0.001**	0.98 (L)
RootsOfUnity	< **0.001**	0.20 (S)	< **0.001**	0.42 (M)	< **0.001**	0.83 (L)
SaddlePointExp.	0.89	0.08	1	0.04	**0.02**	0.27 (S)
Sort	0.84	0.02	0.84	0.04	< **0.001**	0.72 (M)
Tomorrow	< **0.001**	4.73 (L)	0.10	0.18	< **0.001**	4.98 (L)

pared to R-GA and H-OV-GA.

7.6 Threats to validity

This section discusses the threats that could affect the validity of the evaluation of the proposed approach.

Threats to *construct validity* concern the relation between theory and experimentation. In order to evaluate the performance of the experimented techniques we used branch coverage and number of statements executed. These metrics provide a good indication of effectiveness and efficiency in the context of test data generation and are widely adopted in the related literature. Another threat to the construct validity can be related to the manual adjustments performed on the classes for the purpose of the study. However, such changes did not affect the external behavior of the classes and they were performed only because the current version of our prototype does not support non-numeric parameters.

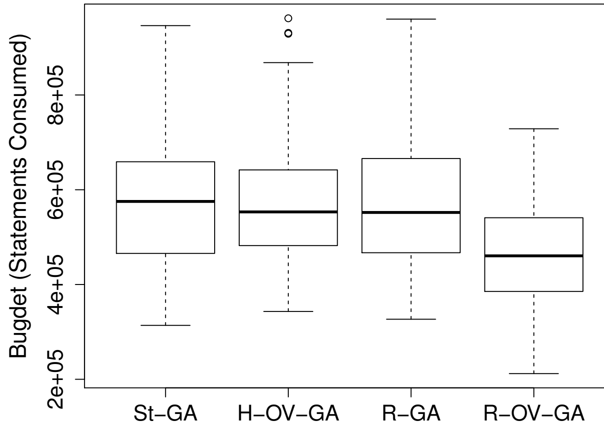


Figure 7.6: Comparison of the budget consumed by St-GA, H-OV-GA, OV-GA and R-GA over 100 independent runs on Sort.

Threats to *internal validity* concern factors that could influence our results. We mitigated the influence of the GA randomness when generating the test suite by repeating the process 100 times and reporting the average performance achieved. Another threat is represented by the approach used to compute orthogonal vectors in the SVD-based GA. In fact, there is no unique way to generate orthogonal vectors and the specific method that is selected might affect the performance of SVD-GA. In order to mitigate such an issue we report—in Section 7.2—the algorithm used to compute the orthogonal column vectors. The setting of the GA parameters represents another threat. To define such parameters we conducted a preliminary set of runs, aimed at assessing the sensitivity of subjects and techniques to the parameters. We observed no major sensitivity. Thus, the detailed configuration reported in this Chapter could be used as a good starting point when using the proposed approach on Java classes having features comparable to those of the classes used in our study.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. In addition to showing values of branch coverage and number of statements executed, we have also statistically compared the effectiveness and efficiency of the experimented methods using the Wilcoxon non-parametric test, indicating whether differences in terms of effectiveness and efficiency are statistically significant. In addition, we also used the Cohen's effect size to estimate the magnitude of the difference.

Threats to *external validity* concern the generalization of our findings. Although we considered data from 17 Java classes extracted from well known and publicly available libraries, the study should be replicated on further programs to corroborate our results. If we consider similar studies published in the related literature, the one presented in this Chapter is comparable or larger.

7.7 Conclusion and future work

Evolutionary test data generation is one of the most widely explored approaches to automate the testing process. Along with its strengths, it has a set of challenges and open issues. One of these issues is the loss of diversity between test cases during the search process for hard-to-cover branches. This chapter proposed the integration of a diversification technique based on SVD-GA and reactive exploration of the search space, with the principal objective of avoiding the search being dominated by a small set of similar individuals. To this aim, this chapter presented three improved variants of the basic GA, which are the history aware SVD-GA, reactive GA, and combined history aware and reactive SVD-GA in the context of evolutionary test data generation.

Experimental studies on 17 Java classes extracted from widely used open source libraries showed strong, statistically significant improvements. The proposed diversification schemes improve over basic GA in terms of *effectiveness*, i.e., branch coverage achieved, and *efficiency*, i.e., search budget consumed. In particular, the combined application of the reactive and SVD-based schemes resulted in the best overall improvement. The combined method significantly improved effectiveness in 8 of the tested classes, while it significantly improved efficiency in 6 out of 7 tested classes where the effectiveness was not significantly improved.

Based on the quite promising results obtained by the empirical study reported in this Chapter, we intend to address a number of remaining issues in our future work. Specifically, we plan to support orthogonal exploration of search spaces with non-numeric data types (e.g. strings) as well as an arbitrary number of method call sequences as part of our future extension. For such an extension, the non-numeric types and the method call sequences need to be mapped to a vector space in which a meaningful notion of evolution direction can be defined. An idea to proceed along these directions involves mapping a variable size sequence (of method calls) to fixed size *feature vectors*, which represent the frequency of occurrence of each feature in an individual. Similarly, feature vectors can be used to represent strings of arbitrary length by extracting pairwise Longest Common Substrings (LCS) from the strings into a feature set and encoding each string as a feature vector containing each feature (LCS). SVD will then be applied to the feature vectors, so as to explore orthogonal regions in the space of feature vectors.

In a similar vein, since the current approach considers vectors of fixed size, we intend to investigate an extension of our technique that is able to address vectors of variable dimensions by using complex number for representing variable length candidate solutions.

Finally, the current prototype implementation is meant only to show the feasibility of the proposed approach and it addresses the issue of diversity in populations from the research perspective. However, to produce a tool that can be used by test engineers for actual testing activities, there are several enhancements that should be made. For instance, incorporating special and boundary values (e.g. *null*, *NaN*, *Infinity*, ...) into the initial population would help to cover trivial error-checking branches, that remain otherwise uncovered. We plan to integrate all our future extensions and enhancements of our implementation into the publicly available tool EvoSuite.

Chapter 8

Multi-Objective Test Suite Optimization

Contents

8.1 Introduction and Motivation **186**

8.2 Injecting Diversity in Multi-Objective Test Suite Optimization: DIV-GA **187**

8.2.1 The DIV-GA algorithm 190

8.3 Empirical Evaluation **191**

8.3.1 Research Questions 192

8.3.2 Test Selection Objectives 193

8.3.3 Experimented Algorithms 195

8.3.4 Evaluation Metrics 197

8.4 Empirical Results **200**

8.4.1 **RQ_1**: *To what extent does DIV-GA produce near optimal solutions, compared to alternative test case selection techniques?* . . . 201

8.4.2 **RQ_2**: *What is the cost-effectiveness of DIV-GA, compared to alternative test case selection techniques?* 208

8.5 Threats to Validity **211**

8.6 Conclusion and Future Work **212**

8.1 Introduction and Motivation

A way to reduce the cost of regression testing consists of selecting or prioritizing subsets of test cases from a test suite according to some criteria. For example, widely used criteria are code coverage [34, 96, 97], program modification [98, 99, 100], execution cost [68, 101, 102], or past fault information [97, 68, 103]. Previous studies have highlighted that when using multiple criteria the optimization of test suite is more effective than when using individual ones [97, 104, 68, 103, 105]. The simplest way to combine different criteria is to conflate all the criteria in a single-objective function to be optimized [34, 96, 101, 102]. Although such an approach is widely used when solving multi-objective optimization problems, this may produce less optimal results compared to Pareto-efficient methods. The test suite optimization problems have been also treated using Pareto-efficient multi-objective genetic algorithms (MOGAs) to deal with multiple and contrasting objectives [68, 103]. However, empirical results indicated that in some cases MOGAs provide better solutions. However, there is no a clear winner between single-objective approaches and MOGAs [68] and their combination is not always useful to achieve better results [103].

The poor performance of MOGAs when applied to test suite optimization can be due to the phenomenon of genetic drift, i.e., a loss of diversity between solutions, which affects not only single-objective GAs but also MOGAs. In such a scenario MOGAs can prematurely converge within some sub-optimal region [109, 345, 106, 346, 136]. Promoting diversity between test cases is a key factor to improve the optimality of GAs [106]. An intuitive strategy to promote diversity consists of adding a diversity-aware fitness function to maximize the diversity with respect to a coverage criterion, as done in our previous work [11] for code coverage. Hemmati *et al.* [291, 310] also suggest to use test case diversity (based on UML state machine coverage) as unique test criterion to be optimized when selecting test cases. However, different coverage criteria might require different diversity-based objective functions, one function for each coverage criterion. Since the performance of MOGAs rapidly decreases for an increasing number of objective functions [321], it is preferable to promote diversity without adding further objective functions. Moreover, the approach used by Hemmati *et al.*[291, 310] is single-objective, and does not provide trade-offs between diversity and test execution cost.

Stemming from these considerations, this Chapter investigates the usage of the SVD-NSGA-II algorithm proposed in Chapter 6 (used for solving multi-objective numerical problems) in the context of test suite optimization problem. Such an algorithm —through the estimation of the evolution directions via Singular Value Decomposition (SVD)— is able to augment the population diversity, by periodically introducing new individuals with orthogonal (unexplored) evolution directions. Specifically, this Chapter presents a variant of the basic SVD-NSGA-II algorithm for binary problems which incorporates a generative algorithm to build a diversified initial population, based on *orthogonal design* [108]. SVD-NSGA-II and orthogonal exploration are two diversity mechanisms that can be applied for any test suite optimization problem and independently of the number of test criteria or objectives to be taken into account.

The usefulness of the proposed diversity mechanisms is analysed through an empirical study conducted on 11 real world open-source programs. Results show that the proposed algorithm outperforms both traditional MOGAs and greedy algorithms. Unlike previous work on multi-criteria regression test case selection [68, 103, 11], which compared meta-heuristics and greedy algorithms only from an optimization point of view, in this Chapter also introduces a performance metric to evaluate the ability of the selected test cases to reveal faults (effectiveness) in a multi-objective paradigm. To the best of knowledge, this is a premier work where for the first time test selection techniques are compared in terms of testing effectiveness. This required the definition of a novel metric to measure the cost-effectiveness of test suite that is inspired by the traditional hypervolume metric widely used for numeric multi-objective problems [110].

Summarising, the contributions of this Chapter are:

1. It introduces a new MOGA, called DIV-GA (DIversity based Genetic Algorithm) which integrates, for the first time, SVD and orthogonal design into MOGAs to solve multi-criteria test case selection problems. The proposed approach addresses the problem of diversity independently of the number and the kind of test criteria.

2. It evaluates DIV-GA on a set of open-source programs extracted from the Siemens suite, the European Space Agency suite and GNU open-source distribution. The selected programs were also used in many previous work [268, 36, 299, 104, 300, 68, 65, 11, 103].

3. It compares DIV-GA with previous techniques: greedy algorithms and NSGA-II, which where used in [68, 65, 11, 103]. The comparison is made by both (i) optimality and (ii) effectiveness point-of-view. To this aim it introduces the cost-effectiveness hypervolume metric I_{CE} to measure the effectiveness of the different algorithms, inspired by the traditional hypervolume metric widely used for numeric multi-objective problems [110].

The Chapter is organized as follows. Section 8.2 presents DIV-GA a variants of SVD-NSGA-II which is combined with the orthogonal design, and describes how to integrate such operators into the main loop of MOGAs. Section 8.3 describes the design of the empirical study conducted to evaluate the benefits of the proposed algorithm. Results are reported and discussed in Section 8.4, while Section 8.5 provides a discussion of the threats that could affect the validity of the results. Section 8.6 concludes the Chapter.

8.2 Injecting Diversity in Multi-Objective Test Suite Optimization: DIV-GA

The SVD-NSGA-II algorithm introduced in Chapter 6 injects diversity by periodically generating new individuals having orthogonal evolution directions that hopefully explore new regions of the search space never explored before. Thus, in this way it prevents the phenomenon of loss of diversity and reduces the probability to prematurely converge toward

Algorithm 11: ORTHOGONAL-POPULATION

Input:
The size of the test suite n; The size of the initial population m;
Result: An initial population P_0 of m individuals.

1 **begin**
2 Generate an Hadamard (square) matrix H_k of size $k = 2^{\lceil \log_2 n \rceil}$, where $\lceil \log_2 n \rceil$ denotes the smallest integer number greater than $\log_2 n$
3 Sort the rows of H_k in ascending order
4 Delete the first column from H_k
5 Select the first m rows and the first n columns from H_k to obtain a new matrix L of size $m \times n$
6 Convert L in a binary matrix $L_m(2^n)$
7 $P_0 \longleftarrow L_m(2^n)$

local optimal Pareto frontier. However, maintaining diversity during the evolution (across generations) is not the only factor which affects the performance of GAs. Indeed, the function used to generate an initial population plays an important role since it performs an initial sampling of the search space [347]. A well-distributed and well-diversified initial population makes the exploration more effective and favors GA convergence toward global optima [347]. Instead, a poorly diversified initial population can compromise the convergence speed and the optimality of the search space because any search algorithm (including SVD-NSGA-II) start its search process in a disadvantaged way. This issue becomes particularly critical for problems where the search space is too large if compared to the size of the population [135]. This is especially true for the test case selection problem, where searching the optimal subset—according to multiple testing criteria—of a test suite of size n requires the are analysis of 2^n possible solutions.

Generally, the initial population is randomly and uniformly generated in the search space. However, for binary problems it is possible to easily generate individuals that are diversified using the concept of orthogonality between binary solutions. According to Zhang *et al.* [348], statistical methods such as *experimental design* can be used to improve the optimality and the convergence speed of GAs. A generic solution of the test suite minimization problem is an array of binary digits $X = \{x_1, \ldots, x_n\}$ where x_i is equal to 1 if the *i-th* test case is selected, 0 otherwise. Then, each element x_i can be considered as a two-level factor[1] which affects the outcome of the fitness function. Hence, the problem of generating a well-distributed initial population for GAs is equivalent to the problem of finding a (small) representative sample of all the possible combinations between factors (test cases) for a given experiment.

Because of such an equivalence, we propose to use the *orthogonal arrays* methodology designed by Montgomery *et al.* [108] for experimental design, to generate an initial population for GAs. Several numeric algorithms can be used to generate such orthogonal arrays, such as

[1] A two-level factor is a factor assuming only two possible values [108]. In our case $x_i \in \{0, 1\}$.

Table 8.1: Orthogonal arrays $L_4(2^3)$ for three test cases where there are four combinations of test cases (factors). The row vectors of such a matrix $L_4(2^3)$ can be used as an initial binary population for GAs.

Combination	Test Case 1	Test Case 2	Test Case 3
1st	0	0	0
2nd	0	1	1
3rd	1	0	1
4th	1	1	0

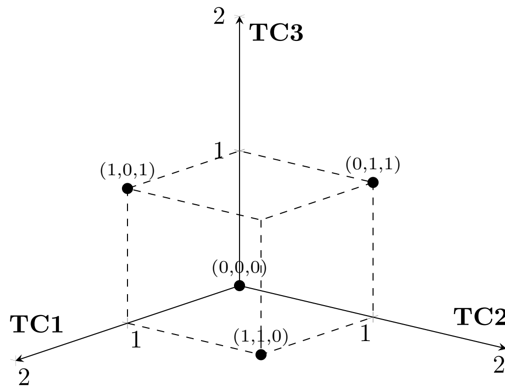


Figure 8.1: Graphical representation of the orthogonal arrays $L_4(2^3)$ for three test cases.

the *row-exchange algorithm* or the *coordinate-exchange algorithm* [108]. They use an iterative search algorithm which incrementally changes the entries of an initial random design matrix X to minimize $\mid (X^T X)^{-1} \mid$. This is equivalent to maximize the determinant of $(X^T X)$, i.e., the differential Shannon entropy of the set of arrays [349].

In this Chapter, we use the Hadamard matrices to build the orthogonal arrays, since such a methodology is particularly efficient for generating two-level orthogonal arrays [350], i.e. the ones required for binary problems such as the test case selection problem. Let N be the size of the test suite and let M be the number of individuals to be generated, we generate the orthogonal arrays (or equivalently the initial population for GAs) using the steps reported in Algorithm 11. Step (2) and step (3) have been implemented using the `hadamard` and the `sortrows` routines, respectively, available in MATLAB [261]. The two steps (4)-(5) simply require to manage the size of the matrix, while step (5) converts the matrix L with values $\in \{-1, 1\}$ into a new matrix $L_m(2^n)$ with values $\in \{0, 1\}$. We propose to use the row vectors of such a matrix $L_m(2^n)$ as individuals of an initial population for GAs, where its generic entry $L_{i,j}$ is equal to 1 if the jth test case is selected by the ith individual, 0 otherwise (step 6 in Algorithm11). Table 8.1 reports an example of a set of four orthogonal arrays generated using the Hadamard matrices for a test suite with three test cases. The orthogonality of such arrays implies that (i) all the test cases occur the same

number of times; (ii) the combinations uniformly cover the whole search space; and (iii) such combinations constitute a set of arrays with minimal mutual information [108]. Hence—as shown in Figure 8.1—the selected combinations are uniformly scattered throughout the space of all possible combinations. The solutions are also well diversified, since the Hamming distance[2] for each pair of solution is equal to 2, which is the maximum distance for sequences of three bits.

As suggested by Zhu *et al.* [351], all these properties make the orthogonal arrays suitable to be used as initial population for GAs, guaranteeing the minimal mutual information between individuals and a scattered uniformly sampling of the search space. It is important to note that the orthogonal arrays used by Zhu *et al.* [351] have more than two-level factors and have been used to solve real-coded numerical problems, while we use two-level orthogonal arrays since test case selection is a multi-objective problem whose solutions are binary arrays. Moreover, we suggest to use a generative approach to build the orthogonal arrays based on Hadamard matrices, which is more efficient for binary populations [350] than the iterative algorithm proposed by Zhu *et al.* [351].

8.2.1 The DIV-GA algorithm

Algorithm 12 reports the novel DIV-GA, the variant of SVD-NSGA-II that integrates the mechanism to promote diversity in the initial population and during the evolution process. First and foremost, in line 3 a uniformly distributed population P_0 is created trough the orthogonal design method, as described in Section 8.2. Then, the main loop of the DIV-GA algorithm first includes k executions of the NSGA-II algorithm, i.e. loop between lines 7–18. During these k generations the usual selection, recombination, and mutation operators are used to create offsprings and the new population is created by selecting the best solutions between parents and offsprings. Then, every k generations we apply our SVD-based preserving technique on the past and current populations P_{old} and P_t respectively.

Finally, the DIV-GA algorithm takes as an input the parameter k, which represents the temporal distance (in terms of number of iterations of the GA) between the two populations on which SVD has to be computed. As shown in [109], the injection of orthogonal individuals through SVD drastically reduces the number of iterations and the total convergence time of a GA, despite the cost of computing SVD. In other words, the lower the value of k, the higher the convergence speed of the GA. Therefore, in principle, SVD could be applied at each iteration ($k = 1$). However, in case the best individuals of two subsequent populations do not change, the SVD might have no effect and then the cost of computing SVD is not compensated by the benefits of injecting orthogonal individuals. In other words, the higher the value of k, the higher the probability that the best individuals of the two populations differ and then the higher the probability that SVD has effect on escaping from local optima. A systematic study on identifying the effects of k on the performances of a GA has not been done yet and is out of the scope of this work. However, previous studies indicate that

[2] The Hamming distance between two binary strings of the same length is equal to the number of positions for which the corresponding binary digits are different.

Algorithm 12: DIV-GA.

Input:
A test suite of size N
Population size M
SVD interval k

```
1  begin
2      t ⟵ 0
3      P_t ⟵ ORTHOGONAL-POPULATION(N,M)
4      old ⟵ t
5      while not (stop condition) do
6          //main loop of NSGA-II
7          Q_t ⟵ MAKE-NEW-POP(P_t)
8          R_t ⟵ P_t ⋃ Q_t
9          𝔽 ⟵ FAST-NONDOMINATED-SORT(R_t)
10         P_{t+1} ⟵ ∅
11         i ⟵ 1
12         while | P_{t+1} | + | 𝔽_i | ⩽ M do
13             CROWDING-DISTANCE-ASSIGNMENT(𝔽_i)
14             P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_i
15             i ⟵ i + 1
16         Sort(𝔽_i) //according to the crowding distance
17         P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_i[1 : (M− | P_{t+1} |)]
18         t ⟵ t + 1
19         if t mod k = 0 then
20             P_t ⟵ INJECT-DIVERSITY(P_old, P_t)
21             old ⟵ t
22     S ⟵ P_t
```

$k = 2$ provides generally good results both in real-coded numerical problems [109] and for evolutionary test data generation [5]. Thus, we also set $k = 2$.

8.3 Empirical Evaluation

The *goal* of this study is to evaluate DIV-GA, with the *purpose* of solving the test case selection problem. The *quality focus* of the study is represented in terms of three—possibly conflicting—objectives which are pursued when performing test case selection, namely increased code coverage capability, decreased execution cost, and increased past fault coverage.

The *context* of the study consists of 11 open-source and industrial programs available from the Software-artifact Infrastructure Repository (SIR) [343]: `space`, an interpreter for Array Description Language, developed by European Space Agency; six GNU open-source programs `bash`, `flex`, `grep`, `gzip`, `sed` and `vim`; four programs of the Siemens suite, namely

Table 8.2: Programs used in the study.

Program	LOCH	# of Test Cases	Description
bash	59,846	1,200	Shell language interpreter
flex	10,459	567	Fast lexical analyser
grep	10,068	808	Regular expression utility
gzip	5,680	215	Data compression program
printtokens	726	4,130	Lexical analyzer
printtokens2	520	4,115	Lexical analyzer
schedule	412	2,650	Priority scheduler
schedule2	374	2,710	Priority scheduler
sed	14,427	360	Non-interactive text editor
space	6,199	13,583	European Space Agency program
vim	122,169	975	Improved *vi* editor

`printtokens`, `printtokens2`, `schedule`, and `schedule2`.

Table 8.2 reports the characteristics of the 11 programs, and specifically their size (in terms of LOCH) and the size of the available test suites (in terms of number of test cases). As it can be noticed, the size of the selected programs ranges between 374 and 122,169 LOCH, while the number of test cases between 215 and 13,583. The selection of these programs was not random. They have been used in previous work on regression testing and especially when experimenting techniques for the selection and prioritization of test suites [268, 104, 65, 68, 11, 103, 36, 299, 300], hence allowing us—wherever possible—to compare results.

In this study, we compare the performance of DIV-GA with two alternative test case selection techniques: (i) one based on additional greedy algorithm used by Rothermel *et al.* [293], and (ii) one based on NSGA-II [107] used by Yoo and Harman [27, 68].

8.3.1 Research Questions

The study aims to provide empirical evidence to answer the following research questions:

- **RQ$_1$**: *To what extent does DIV-GA produce near optimal solutions, compared to alternative test case selection techniques?* This research question aims at evaluating to what extent the proposed DIV-GA algorithm is able to produce a larger number of near optimal solutions, if compared to alternative, state-of-the-art test case selection techniques, namely additional greedy and NSGA-II. Note that, due to the NP-complete nature of the test case selection problem, there is no solver able to find the exact optimal solutions with efficient computational cost, and all the experimented algorithms can provide only near-optimal solutions.

- **RQ$_2$**: *What is the cost-effectiveness of DIV-GA, compared to alternative test case selection techniques?* The purpose of this research question is to evaluate the performance

of DIV-GA—in comparison with alternative techniques—from a cost-effectiveness perspective. This reflect the software engineer's needs to optimize/reduce a test suite without compromising the ability to detect source code defects.

The following subsections describe all the experimented algorithms and the evaluation mechanisms performed to answer the aforementioned research questions.

8.3.2 Test Selection Objectives

We consider three different test case selection criteria, used in previous work [97, 27, 65], namely: (i) statement coverage, (ii) execution cost of test cases, and (iii) past fault coverage.

Statement coverage criterion. We measure statement coverage using the `gcov` tool part of the GNU C compiler (`gcc`). Specifically, we measure statement coverage, as also done by Yoo *et al.* [65]:

$$cov(X) = \frac{1}{m} \sum_{i=1}^{m} \phi_i$$

where m is the total number of code statements to be covered and ϕ_i is equal to 1 if the ith test case is covered by at least one selected test case in X, 0 otherwise.

Execution cost criterion. As for the execution cost, in principle we could just measure the test case execution time. However, performing such a measure is not an easy task, because the measure depends on several external factors such as different hardware, application software, operating system, etc. We address this issue by counting the number of executed elementary instructions in the code, instead of measuring the actual execution time. This is consistent with what was done in previous work on multi-objective test case selection [27, 68]. We use the `gcov` tool to measure the execution frequency of every basic block (elementary instruction) composing each statements. When encountering a function call, `gcov` counts the instructions actually executed when invoking the function; also, complex statements may count as multiple elementary instructions (e.g., the `for` statement counts as three instructions, i.e., the initialization, the condition, and the increment). Hence, the computational cost of each test case is approximated by summing the execution frequencies of all the executed basic blocks. Starting from these execution frequencies, the *execution cost criterion* is defined as follows:

$$cost(X) = \sum_{i=1}^{n} x_i \cdot cost(t_i)$$

where $cost(t_i)$ represents the execution frequency of the ith test case.

Past fault coverage criterion. For the third test criteria, each program has several faulty versions available from the SIR dataset [343]. SIR provides also information about which test cases are able to reveal these faults. Such information can be used to assign a past fault coverage value to each test case subset, computed as the number of known past

faults that this subset is able to reveal in the previous version. Specifically, starting from this history information, the past fault coverage (*fault*) achieved by a solution X can be measured as follows:

$$fault(X) = \frac{1}{m}\sum_{i=1}^{m} f_i$$

where m represents the number of test cases, while f_i is equal to 1 if the ith past fault is covered by at least one selected test case in X, 0 otherwise.

Using the three test case selection criteria described above, we examine two and three-objectives formulations of the test case selection problem. In particular, we consider the two-objective problem taking into account code coverage and execution cost as contrasting goals, similarly to what done in previous work [27, 68, 11]. Formally, the two-objective test case selection problem can be defined as follows:

Problem 6. Two-objective Test Case Selection Problem: *finding a set of optimal solutions X which maximizes the code coverage and minimizes the execution cost:*

$$\max cov(X) = \frac{1}{m}\sum_{i=1}^{m} \phi_i$$
$$\min cost(X) = \sum_{i=1}^{n} x_i \cdot cost(t_i)$$

For the three-objective formulation, we add past fault detection history as a further objective, as also done in previous work [97, 68, 27]. The three-objective test case selection problem can be defined as follows:

Problem 7. Three-objective Test Case Selection Problem: *finding a set of optimal solutions X which maximizes the code coverage, maximizes the past fault coverage and minimizes the execution cost:*

$$\max cov(X) = \frac{1}{m}\sum_{i=1}^{m} \phi_i$$
$$\min cost(X) = \sum_{i=1}^{n} x_i \cdot cost(t_i)$$
$$\max fault(X) = \frac{1}{h}\sum_{i=1}^{h} \varphi_i$$

Note that, besides the test case selection criteria defined above, it is possible to formulate other criteria, e.g., based on data-flow coverage or even functional requirements just providing a clear mapping between tests and criterion-based requirements. However, it is important to highlight that the goal of this work is not to analyse which set of test criteria is the most effective for regression testing. The formulations are used to illustrate how the proposed diversity-preserving technique can be applied to any number and kind of testing criteria to be satisfied.

Algorithm 13: Cost cognizant (or two-objective) Additional Greedy.

```
    Data:
    A program P
    A test suite T = {t_1, ..., t_n}
    Result: A set of sub-test suites S.
1   begin
2       C ⟵ ∅ // covered elements
3       S ⟵ ∅ // selected test cases
4       while C ≤ P do
5           Compute f_i = | S_k − C | / cost_i for each t_i ∈ T
6           t_i ⟵ test case in T with minimum f_i
7           Add t_i to solution C ⟵ C ⋃ t_j
8           T ⟵ T − {t_j}
9           Add S to the Pareto set
```

8.3.3 Experimented Algorithms

For the two-objective formulation of the test case selection problem, we compare the following optimization algorithms:

- The two-objective DIV-GA which uses SVD and orthogonal design to promote diversity between the selected test cases.
- The two-objective *additional greedy* algorithm used by Yoo and Harman [68] and by Rothermel *et al.* [293], which considers at same time both coverage and cost by maximizing the coverage per unit of time of the selected test cases (cost cognizant additional greedy). Algorithm 13 reports its pseudo-code. In particular, let P be a program and T be the set of test cases $t_1, \ldots, t_n$; such an algorithm starts with an empty set of test cases (line 3 of Algorithm 13) and iteratively picks the test case having the best additional coverage per unit cost (lines 5-6 of Algorithm 13). The process ends when the maximum code coverage is reached.
- The two-objective NSGA-II, used by Yoo and Harman [68] for finding a set of non-dominated solutions that maximizes coverage while minimizing the cost of the selected test cases. The NSGA-II pseudo-code is reported in Algorithm 12 in Chapter 2.

Similarly, for what concerns the three-objective formulation of the test case selection problem, we compare the following optimization algorithms:

- The three-objective DIV-GA.
- The three-objective *additional greedy* algorithm used by Yoo and Harman[68], which conflates code coverage, execution cost and past coverage in one objective function to be

Algorithm 14: Three-objectives Additional Greedy.

Data:
A program P
A test suite $T = \{t_1, \ldots, t_n\}$
Result: A set of sub-test suites S.

```
1 begin
2   C ⟵ ∅ // covered elements
3   S ⟵ ∅ // selected test cases
4   while C ≤ P do
5     Compute f_i = (0.5× | S_k − C | +0.5 × fault_i) / cost_i for each t_i ∈ T
6     t_i ⟵ test case in T with minimum f_i
7     Add t_i to solution C ⟵ C ⋃ t_j
8     T ⟵ T − {t_j}
9     Add S to the Pareto set
```

minimized, as highlighted in Algorithm 14. Such an algorithm is similar to Algorithm 13 with the only difference that at each iteration it picks the test case having the best additional code and past faults coverage per unit cost (lines 5-6 of Algorithm 14).

- The three-objective NSGA-II used by Yoo and Harman [68] for finding a set of non-dominated solutions that maximizes code coverage and past fault coverage, while minimizing the cost of the selected test cases.

All the algorithms have been implemented using MATLAB *Global Optimization Toolbox* [261] (release R2011b). In particular, we use the *gamultiobj* routine which implements an island version of NSGA-II algorithm[3]. The DIV-GA algorithm is also implemented by customizing the *gamultiobj* routine, while the computation of the orthogonal arrays[4] is performed using the *rowexch* routine, which implements the *row-exchange algorithm* to generate m orthogonal arrays (m is the size of the initial population) of n factors (test cases) with 2-level ($\{0, 1\}$). For both NSGA-II and DIV-GA we use the same parameters typically used for numerical problems [319]. Specifically:

- **Population size**: since the search space of the test case selection problem is larger for programs with a larger test suite, we use different population sizes according the size of the test suites to be optimized, as shown in Table 8.3.

- **Initial population**: for NSGA-II the initial population is randomly generated within the solution space. For DIV-GA, the initial population is composed of the orthogonal arrays as explained in section 8.2.

[3]The island version of NSGA-II has been used in previous works [103, 103, 11] with the name of *vNSGA-II*.
[4]Orthogonal design used for generating the initial population of DIV-GA.

Table 8.3: Configurations of NSGA-II and DIV-GA for the programs used in the study.

System	Population size	# of generations
bash	400	2,000
flex	400	1,000
grep	200	1,000
gzip	300	500
printtokens	200	500
printtokens2	200	500
schedule	200	500
schedule2	200	500
sed	300	1,000
space	400	1,000
vim	400	2,000

- **Number of generations**: the maximum number of generations varies according to the size of the test suites to be optimized, similarly to what done for the population size. The values are reported in Table 8.3.
- **Crossover function**: we use a multi-point crossover, called *scattered crossover* with probability $p_c = 0.50$.
- **Mutation function**: we use a bit-flip mutation function with probability $p_m = 1/n$, where n is the size of the test suite (or equivalently n is the size of the chromosomes).
- **Stopping criterion**: the average change of the Pareto frontiers is lower than 10^8 for 100 subsequent generations, or the maximum number of generations is reached.
- **SVD frequency**: orthogonal sub-population are generated by SVD every five generation, i.e. $k = 5$. This parameter applies only for DIV-GA.

The setting of GA parameters has been performed using a MATLAB's routine, called *gaoptimset*, which allows to create a list of options to set all parameters. To make faster the execution time of GAs we also used the *vectorized* option of *gaoptimset*, which allows to compute the fitness values for all individuals in the current population at once using a single fitness function call. To this aim we reformulate the computation of the test case selection criteria as matrices multiplications as previously done in [65]. Further details about the formulation of test criteria as matrix operations are reported in Appendix B. Both NSGA-II and DIV-GA have been executed 30 times for each program under study, in order to account for their inherent randomness [265].

8.3.4 Evaluation Metrics

A simple way to evaluate the optimality of a multi-objective optimization algorithm consists of comparing its set of yielded solutions with those of the actual Pareto frontier. However, it

is impossible to know *a priori* the actual Pareto frontier of the test case selection problem, because for large test suites it is unfeasible to compute the global optimal solution using an exhaustive search. In order to perform an *a posteriori* evaluation of the obtained Pareto frontiers, we construct a hybrid frontier by combining the best parts of the different frontiers achieved by all the algorithms (in all the runs), and considering only the solutions that are not dominated by the combined frontier. We call such a hybrid frontier *reference Pareto frontier* [68].

Formally, let $P = \{P_1, \ldots, P_n\}$ be the set of n different Pareto frontiers, the reference Pareto frontier P_{ref} is defined as follows:

$$P_{ref} \subseteq \bigcup_{i=1}^{n} P_i$$

where P_{ref} is the maximal set of non-dominated solutions obtained by all the Pareto frontiers, i.e. $\forall p \in P_{ref} \;\; \nexists q \in P_{ref} : q \succ p$. Thus, P_{ref} helps to compare the global optimality of the different algorithms on the basis of the Pareto frontiers they produce.

We perform a comparison of the different algorithms by estimating two metrics widely used in global optimization problems:

- *Size of Pareto frontier*, that represents the number of non-dominated solutions obtained by each Pareto frontier P_i.
- *Number of non-dominated solutions*, that is the number of solutions that are not dominated by the reference Pareto frontier P_{ref}. Formally, it can be defined as the cardinality of the set $P_i^* = \{p \in P_i : \nexists q \in P_{ref} : q \succ p\}$. In other words, by definition of P_{ref}, P_i^* is the subset of P_i contained in the reference Pareto frontier P_{ref}.

These two metrics were used to answer RQ_1.

We also statistically analyze the obtained results, to check whether the differences between the solutions produced by two different algorithms are statistically significant or not. In particular, the values of the two employed metrics achieved by three algorithms over different independent runs were statistically compared using the *Welch's t test* [226] for both the multi-objective formulations (as done in previous work [68, 11]). Welch's t-test is generally used to test two groups with unequal variance, e.g., in our case the variance of the number of non-dominated solutions produces by the additional greedy and NSGA-II or DIV-GA is different[5]. Significant p-values indicate that the corresponding null hypothesis can be rejected in favor of the alternative hypothesis, i.e., that one of the algorithms produced a Pareto frontier of larger size. In all our statistical tests we reject the null hypotheses for p-values < 0.05 (i.e., we accept a 5% chance of rejecting a null hypothesis when it is true [226]).

We preventively verify the applicability of the Welch's t-test on our data by performing the Wilk-Shapiro normality test. Such a test indicates a non significant deviation from normality

[5]Since the additional greedy is a deterministic algorithm, the variance over 30 independent runs is zero. Conversely, because of the random inheritance of GAs, both NSGA-II and DIV-GA do not reach a zero variance.

(p-value > 0.05). Since we apply the Welch's t-test multiple times, we adjust the p-values using the Holm's correction procedure [227]. This procedure corrects the p-values resulting from n tests by sorting them in ascending order of values and multiplying the smallest by n, the next by $n-1$, and so on.

Other than testing the hypotheses, we also estimated the magnitude of the difference between performances achieved by two algorithms. To this aim, we used the Cohen d effect size [226]. For dependent samples (to be used in the context of paired analyses, as in our study) the Cohen d effect size is defined as the difference between the means (M_1 and M_2), divided by the standard deviation of the (paired) differences between samples (σ_D):

$$d = \frac{M_1 - M_2}{\sigma_D}$$

The effect size is considered *small* for $0.2 \leq \mid d \mid < 0.5$, *medium* for $0.5 \leq \mid d \mid < 0.8$ and *large* for $\mid d \mid \geq 0.8$ [344].

To address (**RQ**$_2$), we analyze the capability of optimized test suites—which may be composed of a smaller number of test cases than the original one—to detect faults. Since each algorithm provides more than one solution—i.e., more than one (near) optimal compromise between cost and coverage—to the best of our knowledge, there is no metric used in previous work to compare the effectiveness of two or more different sub-test suites. A simple way to perform such a comparison consists of plotting the percentage of faults detected by each solution provided by a given algorithm and the corresponding execution cost. This allows to graphically compare two or more Pareto frontiers, showing the percentage of detected faults at same level of execution cost.

In order to quantify the effectiveness of each Pareto frontier, we also use the hypervolume metric, generally used to measure the volume enclosed between a Pareto frontier $P = \{p_1, \ldots, p_n\}$ with respect to an ideal/optimal Pareto frontier R [352, 110]. In other words, the hypervolume measures the closeness of P to the ideal frontier R: the lower the hypervolume value, the better the optimality of the solutions in P [352]. Generally, the hypervolume can be computed within the space of the objectives to be optimized, or using external utility functions selected by the decision maker. In our case, we suggest to revisit the hypervolume metric by taking into account as external utility functions (i) the cost (ii) and the percentage of faults revealed by each solution (sub-set of the test suite) in the Pareto frontier. Hence, the goal is to measure how a Pareto frontier P is optimal from cost and effectiveness point of view. In this context, the ideal frontier R is represented by an ideal set of solutions (sub-test suites) that are able to reveal all the faults for any level of execution cost. According to the proposed utility functions, the weighted hypervolume indicator becomes a bi-dimensional indicator as shown in Figure 8.2.

Without loss of generality, let $P = \{p_1, \ldots, p_n\}$ be a set of solutions, i.e., subsets of test cases. Let $f(p_i)$ be the percentage of faults revealed by the solution $p_i \in P$ and let $cost(p_i)$ be the corresponding execution cost. Let , $R = \{r_1, ..., rn\}$ be the corresponding ideal set of solutions, i.e. subsets of test cases that are able to reveal all faults at varying execution cost

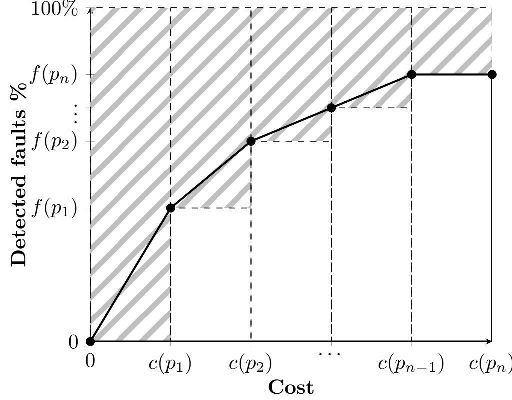


Figure 8.2: Hypervolume metric based on *cost* and *effectiveness* of the sub-test suites.

($cost(r_i) = cost(p_i)$ and $f(r_i) = 1$ for each r_i). The hypervolume enclosed by these points can be easily computed as the sum of rectangles of width $[cost(p_{i+1}) - cost(p_i)]$ and height $[f(r_i) - f(p_i)]$, and then it is equal to:

$$\begin{aligned} I_H(P) &= cost(p_1) + \\ &+ \sum_{i=1}^{n} [cost(p_{i+1}) - cost(p_i)] \cdot [1 - f(p_i)] \end{aligned} \tag{8.1}$$

Figure 8.2 provides a graphical interpretation for the cost-effectiveness hypervolume of a given Pareto frontier. The hypervolume metric is an inverse function: the lower the value, the better the average effectiveness of all the sub-test suites stated in the Pareto frontier[6]. Starting from the cost-effectiveness hypervolume metric, we can express it as percentage of the area (hypervolume) under the ideal/optimal frontier R as follows:

$$I_{CE}(P) = \frac{I_H(P)}{c(p_n)} \tag{8.2}$$

Such a metric measures the (cost-cognizant) weighted average percentage of faults detection loss of a Pareto frontier, i.e., the identified (near) optimal subsets of test suite.

8.4 Empirical Results

This section discusses the results of our study with the aim of answering the research questions formulated in Section 8.3.1.

[6]The hypervolume measures the closeness of the Pareto frontier P to the ideal effectiveness: all the solutions stated in P are able to reveal all the faults.

8.4.1 RQ$_1$: *To what extent does DIV-GA produce near optimal solutions, compared to alternative test case selection techniques?*

Table 8.4 reports the size of Pareto frontiers and the number of non-dominated solutions for the two-objective test case selection problem obtained by (i) DIV-GA, (ii) the additional greedy algorithm, and (iii) NSGA-II. Specifically, the table reports mean size and standard deviation over 30 independent runs of the algorithms. For all the 11 programs, the number of solutions forming the Pareto frontiers of DIV-GA is larger than the number of optimal solutions produced by the NSGA-II and by the additional greedy algorithm. For example, on `gzip` DIV-GA provides a Pareto frontier with 222 solutions on average, while NSGA-II provides 186 solutions and the additional greedy algorithm only 19. For the tester this represents a substantial improvement, since a larger Pareto frontier provides a wider range of candidate solutions that provide different compromises between cost and coverage. From the analysis of Table 8.4 it is also possible to note that there is no clear winner among the additional greedy and NSGA-II, confirming the results of previous study [68]. In 5 cases out of 11, the size of the Pareto frontiers obtained by NSGA-II is smaller than the size of the Pareto frontiers obtained by the additional greedy, while for the remaining cases the scenario is the opposite (the size of the Pareto frontiers achieved by NSGA-II is larger).

DIV-GA also turned out to be able to produce a larger number of non-dominated sub-test suites with respect to all the other algorithms. In particular, the majority of subset of tests forming the Pareto frontiers of DIV-GA are also non-dominated by any other algorithm. For example, on `bash` the Pareto frontier produced by DIV-GA contains 354 solutions, and among them 311 solutions are non-dominated by those obtained by all the other algorithms (i.e., such solutions are stated in the reference Pareto frontiers), while the number of non-dominated solutions provided by the additional greedy algorithm and NSGA-II is really small, especially when compared with the total amount of provided solutions. For example, on `bash` the additional greedy provides 232 different solutions. However, among them only 30 solutions (less than 13%) are non-dominated by the solutions of the other algorithms (in particular by those of DIV-GA). Similarly, NSGA-II obtains 276 different solutions, but only 13% of them are non-dominated by those of the other algorithms. In summary, for the two-objective formulation, DIV-GA not only produces more Pareto optimal sub-test suites than the other algorithms, but they sub-test suites provide better code coverage with lower execution cost than the solutions produced by the other algorithms.

The considerations above are also supported by statistical analysis. Table 8.5 reports the results of the Welch's t-test and Cohen's d effect size, obtained comparing (across the 30 GA runs) the size of the Pareto frontiers and the number of non-dominated solutions achieved by the experimented algorithms (the p-values have been adjusted using the Holm's correction procedure [227]). The statistical analysis confirms that DIV-GA always produces Pareto frontiers that are significantly larger than those produced by the additional greedy algorithm (in 100% of cases) and by NSGA-II (in 82% of cases). The corresponding effect size values revealed that the magnitude of the improvement of DIV-GA over the other two algorithms

Table 8.4: Two-objective test case selection: mean Pareto sizes and number of non-dominated solutions achieved by the different algorithms.

Program	**Method**	**Pareto size**		**Non Dominated Solutions**	
		Mean	**St. Dev.**	**Mean**	**St. Dev.**
bash	DIV-GA	**354**	4.98	**311**	35.39
	Add. Greedy	232	-	30.20	27.24
	NSGA-II	276	5.69	37	52.04
flex	DIV-GA	**306**	8.52	**303**	8.23
	Add. Greedy	43	-	7	0
	NSGA-II	157	70.99	0	-
grep	DIV-GA	**162**	2.19	**140**	3.05
	Add. Greedy	70	-	9	-
	NSGA-II	60	-	3.75	4.79
gzip	DIV-GA	**222**	3.71	**186**	13.29
	Add. Greedy	19	-	5.67	-
	NSGA-II	88	1.36	30	13.48
sed	DIV-GA	**270**	-	**252**	18.03
	Add. Greedy	33	-	3	-
	NSGA-II	225	33.72	27.45	39.71
printtokens	DIV-GA	**86**	5.26	**55.64**	11.75
	Add. Greedy	10	-	3	-
	NSGA-II	6.60	2.30	0	-
printtokens2	DIV-GA	**96**	11.31	**63**	18.57
	Add. Greedy	10	-	4	-
	NSGA-II	23.60	7.50	0	-
schedule	DIV-GA	**62.22**	3.03	**28.33**	2.69
	Add. Greedy	6	-	2	-
	NSGA-II	1	-	0	-
schedule2	DIV-GA	**63.22**	5.49	**31.14**	4.45
	Add. Greedy	9	-	4	-
	NSGA-II	18	0.58	0	-
space	DIV-GA	**344**	4.19	**340**	12.39
	Add. Greedy	119	-	3	-
	NSGA-II	284	85.58	6.67	14.38
vim	DIV-GA	**353**	1.10	**234**	75.37
	Add. Greedy	266	-	91	77.78
	NSGA-II	339	8.66	6.5	9.19

is large (d >1) in the majority of cases: 100% of the cases for the additional greedy and 91% of the cases for NSGA-II.

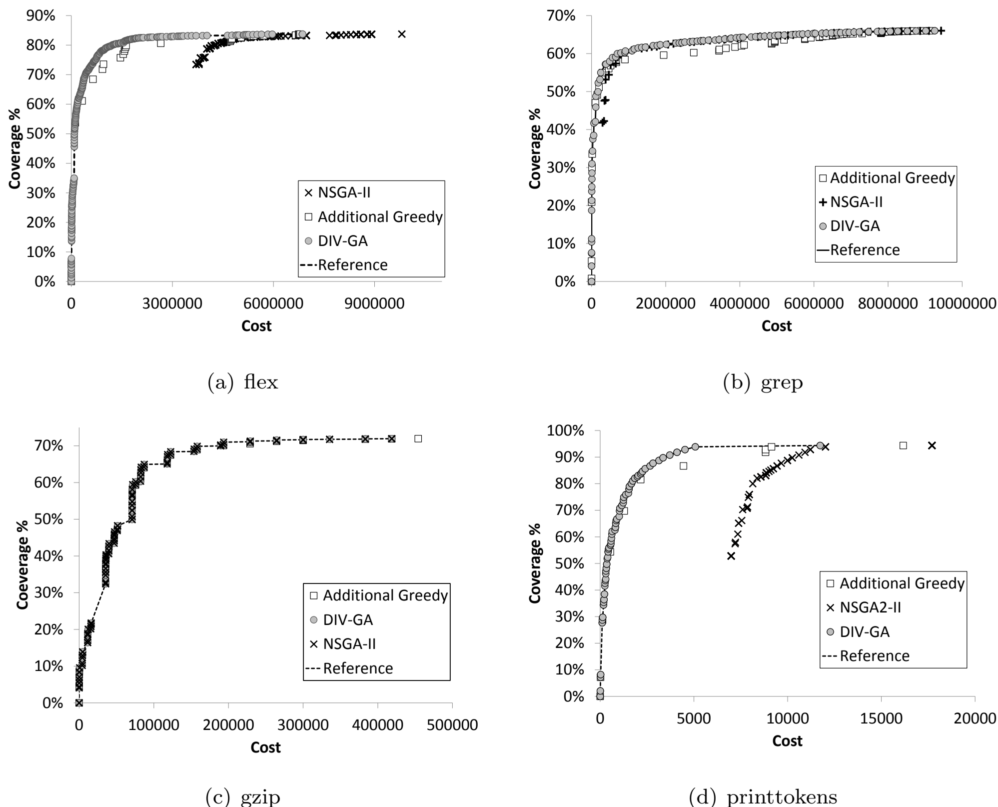
As for the number of solutions DIV-GA statistically outperforms the other two algorithms with a large effect size in all the cases. When comparing the additional greedy and NSGA-II, we can also note that for all the 11 programs none of the two algorithms turns out to be statistically better than the other in terms of Pareto optimality. In fact, in some cases the additional greedy significantly outperforms NSGA-II, while in the other cases it is the

Table 8.5: Comparison between different algorithms for the two-objective test case selection problem. Welch's t-test p values and Cohen's d effect size. We use S, M, and L to indicate small, medium and large effect sizes, respectively.

Program	Hypothesis			Pareto Size		Non Dom. Solutions	
				p-values	**Cohen's d**	**p-values**	**Cohen's d**
bash	DIV-GA	>	Add. Greedy	< **0.01**	58.39 (L)	< **0.01**	8.90 (L)
	DIV-GA	>	NSGA-II	< **0.01**	16.64 (L)	< **0.01**	6.16 (L)
	Add. Greedy	>	NSGA-II	1	-10.96 (L)	0.58	0.16
flex	DIV-GA	>	Add. Greedy	< **0.01**	40.92 (L)	< **0.01**	50.96 (L)
	DIV-GA	>	NSGA-II	0.75	2.94 (L)	< **0.01**	52.69 (L)
	Add. Greedy	>	NSGA-II	1	-2.04 (L)	< **0.01**	-
grep	DIV-GA	>	Add. Greedy	< **0.01**	59.13 (L)	< **0.01**	56.11 (L)
	DIV-GA	>	NSGA-II	< **0.01**	12.37 (L)	< **0.01**	34.05 (L)
	Add. Greedy	>	NSGA-II	< **0.01**	2.30 (L)	< **0.01**	1.44 (L)
gzip	DIV-GA	>	Add. Greedy	< **0.01**	77.31 (L)	< **0.01**	19.19 (L)
	DIV-GA	>	NSGA-II	< **0.01**	47.92 (L)	< **0.01**	11.67 (L)
	Add. Greedy	>	NSGA-II	1	-71.41 (L)	1	-2.56 (L)
printtokens	DIV-GA	>	Add. Greedy	< **0.01**	19.47 (L)	< **0.01**	6.39 (L)
	DIV-GA	>	NSGA-II	< **0.01**	18.59 (L)	< **0.01**	6.65 (L)
	Add. Greedy	>	NSGA-II	< **0.01**	1.68 (L)	< **0.01**	4.32 (L)
printtokens2	DIV-GA	>	Add. Greedy	< **0.01**	10.71 (L)	< **0.01**	2.86 (L)
	DIV-GA	>	NSGA-II	< **0.01**	6.39 (L)	< **0.01**	3.04 (L)
	Add. Greedy	>	NSGA-II	0.41	0.14 (L)	< **0.01**	14.32 (L)
schedule	DIV-GA	>	Add. Greedy	< **0.01**	26.22 (L)	< **0.01**	13.83 (L)
	DIV-GA	>	NSGA-II	< **0.01**	28.33 (L)	< **0.01**	14.19 (L)
	Add. Greedy	>	NSGA-II	0.78	20.74 (L)	**0.02**	3.77 (L)
schedule2	DIV-GA	>	Add. Greedy	< **0.01**	16.95 (L)	< **0.01**	8.44 (L)
	DIV-GA	>	NSGA-II	< **0.01**	11.49 (L)	< **0.01**	9.66 (L)
	Add. Greedy	>	NSGA-II	1	-25.56 (L)	**0.03**	4.54 (L)
sed	DIV-GA	>	Add. Greedy	< **0.01**	70.57 (L)	< **0.01**	19.52 (L)
	DIV-GA	>	NSGA-II	< **0.01**	3.10 (L)	0.14	7.28 (L)
	Add. Greedy	>	NSGA-II	1	-11.06 (L)	0.97	-0.88 (L)
space	DIV-GA	>	Add. Greedy	< **0.01**	76.12 (L)	< **0.01**	38.41 (L)
	DIV-GA	>	NSGA-II	**0.05**	1.00 (L)	< **0.01**	26.22 (L)
	Add. Greedy	>	NSGA-II	1	2.72 (L)	0.69	-0.22 (S)
vim	DIV-GA	>	Add. Greedy	< **0.01**	12.57 (L)	< **0.01**	1.86 (L)
	DIV-GA	>	NSGA-II	< **0.01**	4.18 (L)	< **0.01**	4.17 (L)
	Add. Greedy	>	NSGA-II	1	17.50(L)	< **0.01**	1.45 (L)

contrary.

Figure 8.3 provides—for some programs considered in our study, i.e., `flex`, `grep`, `gzip` and `printtokens`—a graphical comparison between the Pareto frontiers obtained by the three algorithms and a "reference" Pareto frontier, built as explained in section 8.3.4. We obtained consistent results for all other programs as well (see the Appendix C for further details). As it can be noticed, the Pareto frontiers provided by DIV-GA are much closer to the reference Pareto frontiers (often the two frontiers are perfectly overlapped) than the Pareto frontiers provided by NSGA-II and the additional greedy. The additional greedy algorithm provides solutions that are, in some cases, quite close to the reference frontiers. However, the majority of them are dominated by solutions produced by DIV-GA. Instead, NSGA-II produces solutions quite far from the optimal set of solutions (e.g., on `printtokens`). Only on `gzip` all the algorithms turned out to be close to the reference frontier, even if Table 8.4 shows a substantial difference in terms of number of non-dominated solutions and Pareto



(a) flex (b) grep

(c) gzip (d) printtokens

Figure 8.3: Pareto frontiers.

frontier size of DIV-GA compared to the other approaches. Particularly interesting are the results achieved for all the programs in the Siemens suite —i.e. `printtokens`, `printtokens2`, `schedule`, and `schedule2`— where NSGA-II is very far from the optimal Pareto frontier while DIV-GA provides (near) optimal frontiers. Finally, for what concerns the uniformity of the distribution of the solutions over the produced Pareto frontiers, we can also observe that DIV-GA also provides a wider diversity of non-dominated solutions with higher coverage and uniformity along the Pareto frontier than the other algorithms. Conversely, NSGA-II and the additional greedy algorithm provide solutions which are not well distributed uniformly along the Pareto frontiers, i.e., providing more solutions for higher coverage levels and leaving the rest of the Pareto frontiers quite unexplored.

Table 8.6 compares the performance of the three experimented algorithms for the three-objective formulation of the test case selection problem. Also in this case, results are shown in terms of mean and standard deviation of the size of Pareto frontiers and number of non-dominated solutions computed across 30 independent runs of each algorithm. In all cases, the size of the Pareto frontiers obtained by DIV-GA is larger than those obtained using

Table 8.6: Three-objective formulation of the test case selection problem: mean size of Pareto frontier and mean number of non-dominated solutions obtained by the different algorithms.

Program	**Method**	**Pareto size**		**Non Dominated Solutions**	
		Mean	**St. Dev.**	**Mean**	**St. Dev.**
bash	DIV-GA	**354**	2.98	**310**	53.73
	Add. Greedy	233	-	31	-
	NSGA-II	276	4.47	48	69.20
flex	DIV-GA	**331**	8.06	**328**	8.04
	Add. Greedy	47	-	7	-
	NSGA-II	139	44.62	0	-
grep	DIV-GA	**317**	13.06	**294**	44.21
	Add. Greedy	72	-	6	-
	NSGA-II	207	46.23	21	42.73
gzip	DIV-GA	**198**	4.97	**171**	11.30
	Add. Greedy	19	-	1	-
	NSGA-II	**195**	4.66	130	20.77
printtokens	DIV-GA	**110**	33.28	**110**	33.71
	Add. Greedy	13	-	7	-
	NSGA-II	6	2.77	0	-
printtokens2	DIV-GA	**190**	30.11	**189**	27.70
	Add. Greedy	11	-	4	-
	NSGA-II	11	3.50	0	-
schedule	DIV-GA	**213**	11.33	**199**	11.29
	Add. Greedy	10	-	6	-
	NSGA-II	123	21.48	18.67	30.24
schedule2	DIV-GA	**136**	23.09	**91**	27.77
	Add. Greedy	11	-	1	-
	NSGA-II	118	20.47	9	20.20
sed	DIV-GA	**195**	38.02	**164**	43.48
	Add. Greedy	33	-	5	-
	NSGA-II	98	19.83	36.14	12.56
space	DIV-GA	**360**	-	**318**	59.51
	Add. Greedy	126	-	7	-
	NSGA-II	**360**	-	78	91.65
vim	DIV-GA	**393**	**1.77**	**219**	**16.29**
	Add. Greedy	266	-	163	-
	NSGA-II	182	3.50	26.43	10.75

the additional greedy algorithm. For example, on `gzip` DIV-GA provides 198 solutions against 19 solutions produced using the additional greedy. DIV-GA also produces larger Pareto frontiers than NSGA-II in 9 out of 11 cases, while on `space` and `gzip` the size of the Pareto frontiers is exactly the same. However, by looking at the non-dominance of the solutions, DIV-GA always produces a number of non-dominated solutions larger than the number of non-dominated solutions produced by the other two algorithms. Such solutions are also stated on the reference frontier, i.e., they are not dominated by solutions produced

Table 8.7: Comparison between different algorithms for the three-objective test case selection problem. Welch's t-test p values and Cohen's d effect size. We use S, M, and L to indicate small, medium and large effect sizes respectively.

Program	Hypothesis			Pareto Size		Non Dom. Solutions	
				p-values	**Cohen's d**	**p-values**	**Cohen's d**
bash	DIV-GA	>	Add. Greedy	< **0.01**	57.48 (L)	< **0.01**	7.31 (L)
	DIV-GA	>	NSGA-II	< **0.01**	20.59 (L)	< **0.01**	4.22 (L)
	Add. Greedy	>	NSGA-II	1	-13.60 (L)	0.73	0.34 (S)
flex	DIV-GA	>	Add. Greedy	< **0.01**	49.82 (L)	< **0.01**	56.40 (L)
	DIV-GA	>	NSGA-II	< **0.01**	5.96 (L)	< **0.01**	57.49 (L)
	Add. Greedy	>	NSGA-II	1	-2.94 (L)	< **0.01**	20.62
grep	DIV-GA	>	Add. Greedy	< **0.01**	26.56 (L)	< **0.01**	9.21 (L)
	DIV-GA	>	NSGA-II	< **0.01**	3.25 (L)	< **0.01**	6.28 (L)
	Add. Greedy	>	NSGA-II	1	-4.12 (L)	< **0.01**	0.49 (M)
gzip	DIV-GA	>	Add. Greedy	< **0.01**	50.88 (L)	< **0.01**	56.00 (L)
	DIV-GA	>	NSGA-II	0.25	0.62 (M)	< **0.01**	4.46 (L)
	Add. Greedy	>	NSGA-II	1	-53.37 (L)	1	-8.81 (L)
printtokens	DIV-GA	>	Add. Greedy	< **0.01**	4.14 (L)	< **0.01**	4.39 (L)
	DIV-GA	>	NSGA-II	< **0.01**	4.43 (L)	< **0.01**	4.66 (L)
	Add. Greedy	>	NSGA-II	< **0.01**	3.68 (L)	< **0.01**	19.45 (L)
printtokens2	DIV-GA	>	Add. Greedy	< **0.01**	8.41 (L)	< **0.01**	9.42 (L)
	DIV-GA	>	NSGA-II	< **0.01**	8.377 (L)	< **0.01**	9.62 (L)
	Add. Greedy	>	NSGA-II	0.40	0.12	< **0.01**	12.13 (L)
schedule	DIV-GA	>	Add. Greedy	< **0.01**	9.12 (L)	< **0.01**	24.21 (L)
	DIV-GA	>	NSGA-II	< **0.01**	2.63 (L)	< **0.01**	7.91 (L)
	Add. Greedy	>	NSGA-II	1	-7.46 (L)	0.82	0.59 (M)
schedule2	DIV-GA	>	Add. Greedy	< **0.01**	7.63 (L)	< **0.01**	4.57 (L)
	DIV-GA	>	NSGA-II	0.14	0.80 (L)	< **0.01**	3.38 (L)
	Add. Greedy	>	NSGA-II	1	-7.39 (L)	0.94	0.55 (M)
sed	DIV-GA	>	Add. Greedy	< **0.01**	6.03 (L)	< **0.01**	7.07 (L)
	DIV-GA	>	NSGA-II	< **0.01**	3.19 (L)	< **0.01**	5.62 (L)
	Add. Greedy	>	NSGA-II	1	-4.65 (L)	1	3.51 (L)
space	DIV-GA	>	Add. Greedy	< **0.01**	251.46 (L)	< **0.01**	8.98 (L)
	DIV-GA	>	NSGA-II	< **0.01**	145.73 (L)	< **0.01**	1.02 (L)
	Add. Greedy	>	NSGA-II	1	-83.09 (L)	0.93	-2.64 (L)
vim	DIV-GA	>	Add. Greedy	< **0.01**	106.25 (L)	< **0.01**	4.88 (L)
	DIV-GA	>	NSGA-II	< **0.01**	76.70 (L)	< **0.01**	13.97 (L)
	Add. Greedy	>	NSGA-II	< **0.01**	33.84 (L)	< **0.01**	17.96 (L)

by the other algorithms. Conversely, the majority of solutions produced by the additional greedy and NSGA-II are dominated by—i.e., they are worse than—the solutions of DIV-GA. For example, on `flex` the additional greedy algorithm produces 47 solutions forming the Pareto frontier, and among them only 7 solutions (less than 15% of the total amount of the produced solutions) are non-dominated by any other algorithm. Similarly, NSGA-II produces 139 solutions for the same program, but none of them belongs to the reference Pareto frontier, i.e., all of them are dominated by other solutions. Hence, for the three-objective test case selection problem, DIV-GA produces a larger number of sub-test suites with higher code coverage, higher past fault coverage and lower execution cost than both NSGA-II and the additional greedy.

Results of the Welch's t test confirm that the differences between DIV-GA and the other two algorithms are also statistically significant. Table 8.7 reports the p-values obtained comparing the Pareto frontier size and the number of non-dominated solutions achieved

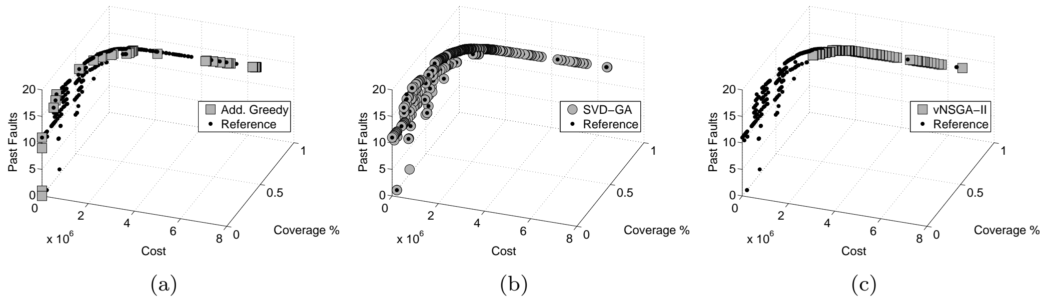


Figure 8.4: Three-objective Pareto Frontiers achieved on *flex*.

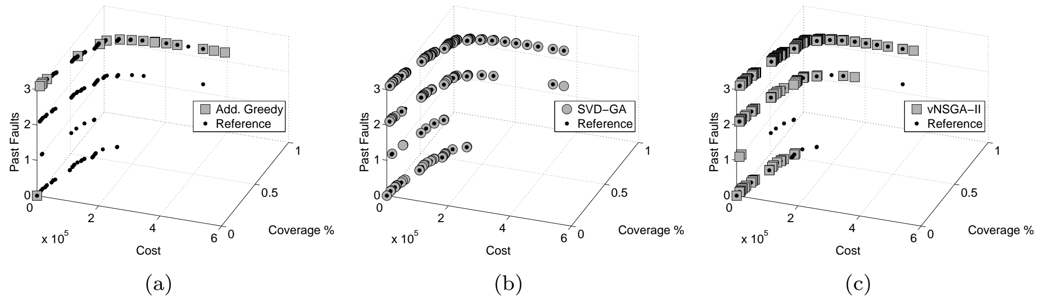


Figure 8.5: Three-objective Pareto Frontiers achieved on *gzip*.

by the experimented algorithms (the p-values have been adjusted using the Holm's [227] correction procedure). For all programs, the Pareto frontiers produced by DIV-GA are significantly larger than those produced by the additional greedy (100% of cases) and by NSGA-II (82% of cases) with a very large effect size in all the cases. When comparing NSGA-II and the additional greedy, we can also note that NSGA-II produces a larger number of sub-test suites than the additional greedy. DIV-GA also always produces a larger number of solutions—i.e., more sub-test suites—that are non-dominated by any solution obtained by the other algorithms. The results also show that in general the additional greedy algorithm is dominated by NSGA-II. However, on some programs —i.e. `flex`, `grep`, `printtokens`, `printokens2`, and `vim`— the additional greedy algorithm produces more optimal results than NSGA-II, as it was also pointed out by Yoo *et al.* [68].

Figures C.5-C.8 show the results for the three-objective formulation on three programs , i.e., `flex`, `gzip`, and `printtokens`. Consistent results have been obtained for all the other programs (see the Appendix C for further details). The 3D plots displays the solutions produced by (i) DIV-GA, (ii) the additional greedy algorithm, (iii) NSGA-II, and (iv) the reference Pareto frontier (denoted using black dots). The additional greedy algorithm produces solutions that are quite close to the reference frontier. However, the number of the

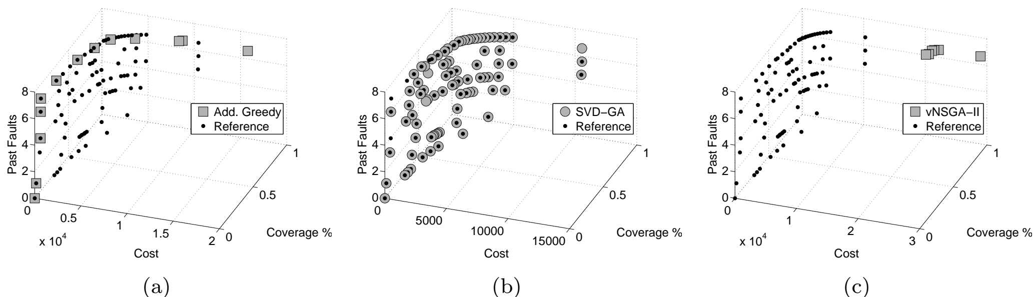


Figure 8.6: Three-objective Pareto Frontiers achieved on *printtokens*.

produced solutions is really small if compared to the reference frontier. DIV-GA always produces three-objective solutions stated in the reference frontier: in all cases the Pareto frontier of DIV-GA exactly overlaps the reference frontier. Hence, DIV-GA always produces solutions that are non-dominated by any other algorithm. Instead, NSGA-II produces (near) optimal solutions only in a few cases. For example, on `gzip` the Pareto frontier obtained by NSGA-II is quite close to the reference Pareto frontier, even if Table 8.7 reveals that only a part of such solutions are non-dominated, while on `printtokens`, the solutions obtained by NSGA-II are quite far from the reference Pareto frontier.

RQ1 Summary. *For both the two- and three-objective test case selection problems, we can conclude that DIV-GA is always able to produce more Pareto-optimal sub-test suites than the additional greedy algorithm and NSGA-II. Such sub-test suites represent Pareto-optimal compromises between coverage and cost.*

8.4.2 RQ$_2$: *What is the cost-effectiveness of DIV-GA, compared to alternative test case selection techniques?*

Table 8.8 reports the mean values of the cost-effectiveness hypervolume metric (I_{EC}) related to the different Pareto frontiers produced by the three experimented algorithms: (i) DIV-GA, (ii) additional greedy, and (iii) NSGA-II. The reported values represent the mean of the I_{EC} values achieved over 30 independent runs. Results are also collected according to the two formulations of test case selection problem investigated in this Chapter.

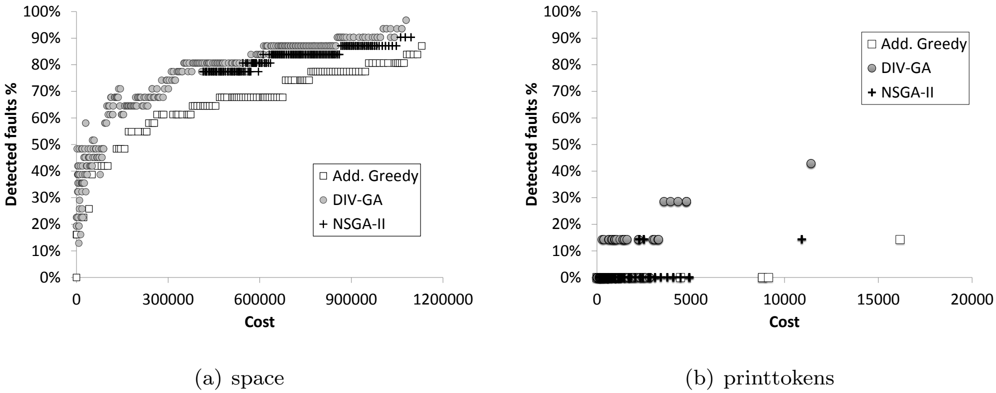
For the two-objective formulation, in 10 out of 11 programs the hypervolume values obtained by DIV-GA are smaller than those achieved by the additional greedy. Hence, the test cases stated in the corresponding Pareto frontiers are able to detect more faults with a lower execution cost (which mirrors a lower average percentage of fault detection loss). Only on `sed`, the additional greedy produces the same hypervolume values as DIV-GA. A similar analysis can be done by comparing DIV-GA and NSGA-II: for all programs, the I_{EC} values provided by DIV-GA are better than those provided by NSGA-II. When comparing the additional greedy and NSGA-II, it is possible to observe that there is no clear winner

Table 8.8: Mean cost-effectiveness hypervolume values.

Program	Method	I_{CE}	
		2-Objective	**3-Objective**
bash	Add. Greedy	0.45	0.32
	DIV-GA	**0.39**	**0.30**
	NSGA-II	0.52	0.48
flex	Add. Greedy	0.03	0.15
	DIV-GA	**0.02**	**0.05**
	NSGA-II	0.04	0.20
grep	Add. Greedy	0.51	0.51
	DIV-GA	**0.40**	**0.27**
	NSGA-II	**0.40**	0.28
gzip	Add. Greedy	0.56	0.52
	DIV-GA	**0.51**	**0.51**
	NSGA-II	0.60	0.54
printtokens	Add. Greedy	1	0.72
	DIV-GA	**0.83**	**0.60**
	NSGA-II	0.97	0.81
printtokens2	Add. Greedy	0.86	0.17
	DIV-GA	**0.81**	**0.08**
	NSGA-II	1	**0.86**
schedule	Add. Greedy	1	0.07
	DIV-GA	**0.97**	**0.06**
	NSGA-II	0.98	0.10
schedule2	Add. Greedy	1	0.89
	DIV-GA	**0.99**	**0.84**
	NSGA-II	1	0.89
sed	Add. Greedy	**0.23**	0.20
	DIV-GA	**0.23**	**0.10**
	NSGA-II	0.28	**0.10**
space	Add. Greedy	0.35	0.14
	DIV-GA	**0.24**	**0.12**
	NSGA-II	0.35	0.16
vim	Add. Greedy	0.24	0.23
	DIV-GA	**0.22**	**0.19**
	NSGA-II	0.33	0.25

among them, also from an effectiveness point of view. In 2 out of 11 cases, the test cases selected by NSGA-II can detect more faults than the solutions detected by the additional greedy algorithm, while in 6 out of 11 cases the greedy algorithm outperforms NSGA-II. In the remaining 3 cases, there is no difference between the I_{EC} values achieved by the two algorithms. This result indicates that *injecting diversity* in GAs allows to improve the effectiveness of multi-objective GAs (NSGA-II in our case).

To provide a graphical interpretation to the I_{EC} metric, Figure C.15 plots the percentage of faults detected by the solutions (sub-test suites) provided by the three algorithms at same



(a) space

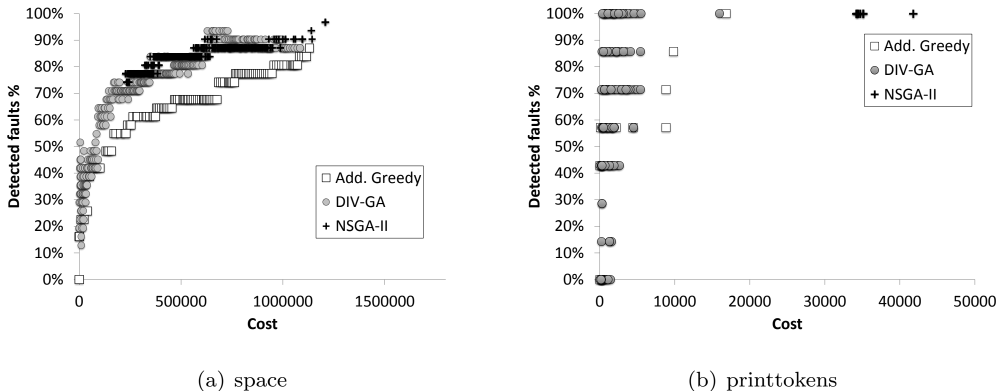
(b) printtokens

Figure 8.7: Effectiveness of the achieved sub-test suites for two-objective test case selection.

level of execution cost on `space` and `printtokens`. We can observe that the sub-test suites selected by DIV-GA are able to detect more faults than the additional greedy with lower execution cost. For example, on `space` the test suites optimized by DIV-GA can detect 100% of faults, while the percentage of faults produced by the other two algorithms is lower at the same level of execution cost. Similar considerations can be made on the other programs (see the Appendix C for further details).

For the three-objective formulation of the test case selection problem, we obtained results that are quite similar to those obtained for the two-objective formulation (see Table 8.8). Indeed, for all the programs, the cost-effectiveness hypervolume values achieved by DIV-GA are smaller than those achieved by the additional greedy. This means that the test cases within the corresponding Pareto frontiers are able to detect more faults with a lower execution cost, mirroring a lower average percentage of fault detection loss per unit cost. A similar analysis can be done by comparing DIV-GA and NSGA-II. In general, DIV-GA obtains the best hypervolume values. Only on `grep` and `sed` the I_{CE} values obtained by DIV-GA and NSGA-II are the same. When comparing the additional greedy and NSGA-II, we can observe that in general the additional greedy is better in terms of fault detection-effectiveness. In 2 out of 11 cases, the test cases selected by NSGA-II can detect more faults than the test cases selected by the additional greedy algorithm, while in 8 out of 11 cases the additional greedy outperforms NSGA-II. Finally, in only one case there is no difference between the I_{EC} values produced by the two algorithms.

Figure C.16 plots the cost/faults curves obtained by the three algorithms. The goal of such an analysis is to provide a graphical comparison of the percentage of faults detected by the different solutions (sub-test suites) at same level of execution cost. We can notice that the sub-test suites obtained by DIV-GA are able to detect more solutions than both the additional greedy and NSGA-II with a lower (or in some cases the same) execution cost. For example, on `space`, the test suites optimized by DIV-GA can detect 100% of the faults,



(a) space

(b) printtokens

Figure 8.8: Effectiveness of the achieved sub-test suites for three-objectives test case selection.

while the percentage of faults detected by the other two algorithms is lower for the same level of execution cost. On `printtokens`, all the algorithms provide solution that are able to reveal all faults. However, DIV-GA turned out to be better than the other techniques in terms of execution cost. Similar considerations can be made on the other programs (see the Appendix C for further details).

RQ$_2$ Summary. *For both two and three-objective formulations of the test case selection problem, we can conclude that DIV-GA is always able to produce optimal sub-test suites within the Pareto frontiers. Such sub-test suites are able to reveal more faults than the sub-test suites obtained by the additional greedy algorithm and NSGA-II. Moreover, the corresponding execution cost is lower than the other techniques.*

8.5 Threats to Validity

This section discusses the threats to the validity of our empirical evaluation, classifying them into *construct*, *internal*, *external*, and *conclusion* validity.

Threats to *construct validity* concern the relationship between theory and observation. In this study, they are mainly related to the choice of the metrics used to evaluate the characteristics of the different test case selection algorithms. In order to evaluate the optimality of the experimented algorithms (DIV-GA, additional greedy, and NSGA-II) we used two well-known metrics: (i) Pareto frontier size and (ii) number of solutions not dominated by reference Pareto frontiers [319]. Such metrics have been also used in previous work on multi-objective test case selection [68, 103, 11, 304]. The effectiveness of the Pareto sets achieved by all the algorithms was measured by using the hypervolume indicator, which is widely used in multi-objective optimization [110]. In particular, we used (i) execution cost and (ii) percentage of detected faults as utility functions to build a bi-objective hypervolume indicator. Another construct validity threat involves the correctness of the measures used

as test criteria: statement coverage, faults coverage and execution cost. To mitigate such a threat, the code coverage information were collected using two open-source profiler/compiler tools (GNU `gcc` and `gcov`). The execution cost was measured by counting the number of source code statements expected to be executed by the test cases, while the original fault coverage information was extracted from the SIR dataset [343].

Threats to *internal validity* concern factors that could have influenced our results and that were not properly considered. In this study, a crucial factor is the random nature of the GAs themselves [265]. To address this problem, we ran the experimented GAs (DIV-GA and NSGA-II) 30 times for each subject program (as done in previous work [68, 65, 11]), and considered the mean values of the measures used to evaluate the optimality and effectiveness. Another threat to internal validity is represented by the algorithms used to compute orthogonal vectors in the DIV-GA: there is no unique algorithm to generate orthogonal vectors and different algorithms might affect the performance of the proposed algorithm. To address such a potential issue we report the algorithm used in this Chapter. The tuning of the GAs parameters is another factor that can affect the internal validity of this work. In this study we used the same parameters used in previous work on multi-objective test case selection [68, 65, 11].

Threats to *external validity* concern the generalization of our findings, and are related to the set of programs used in the experimentation. We considered 11 programs extracted from the SIR, that were also used in most previous work on regression testing [268, 36, 299, 104, 300, 68, 65, 11, 103]. However, in order to corroborate our findings, replications on a wider range of programs and optimization techniques are desirable. The replication of the study we conducted in this Chapter is part of our agenda for future work. Also, there may be optimization algorithms or formulations of the test case selection problem not considered in this study that could produce better results. No particular algorithm is known to be effective for the multi-objective test case selection problem [103], and usually the evaluation of a search-based algorithm involves a comparison with other kinds of algorithms. We compared DIV-GA with (i) the additional greedy algorithm, and (ii) the Pareto efficient multi-objective GA (NSGA-II) in order to evaluate the benefits of the proposed algorithms over the most used ones. Moreover, in order to make more generalizable the results, we evaluated all the algorithms with respect to solving two different formulations of the test case selection problem with two and three objectives to be optimized.

Finally, for what concerns *conclusion validity*, we support our findings by using appropriate statistical tests, i.e. the Welch's t-test. We performed Wilk-Shapiro normality test to verify wether the Welch's t-test could be applied to our data. Finally, we used the Cohen's d effect size to measure the magnitude of the differences between the experimented algorithms.

8.6 Conclusion and Future Work

We proposed a novel diversity preserving technique based on Singular Value Decomposition and orthogonal arrays to improve the performance of multi-objective genetic algorithms when

solving multi-criteria regression testing problems. Specifically, we proposed DIV-GA (DIversity based Genetic Algorithm), a novel multi-objective genetic algorithm which combine the main loop of the popular NSGA-II with the diversity preserving mechanism formulated for multi-objective test case selection.

An empirical study conducted on 11 open source programs and test suites shows that DIV-GA outperforms both additional greedy algorithm and NSGA-II, which were considered as the best optimizers for multi-objective test case selection problem [68, 27, 65, 34, 35, 36]. In particular, DIV-GA allows not only to generate more optimal trade-offs with respect to the other optimizers when considering two and three test case selection criteria, but its selected sub-test suites turned out to be more cost-effective. Indeed, the sub-test suites generated by DIV-GA are able to reveal more faults at same level of execution cost than the sub-test suites obtained by both the additional greedy algorithm and NSGA-II.

The results achieved in our experimentation support our initial conjecture. Preserving diversity is a crucial activity when using search-based algorithms for test case selection. Without sound diversity mechanisms, which might require using approaches such as the one proposed in this Chapter, the potential of search-based algorithms can be seriously undermined, as shown in our empirical study. This is true in the test case selection problem, but also in other software engineering problems, such as test case generation [5]. Our final conjecture, that we plan to verify in the future, is that the proposed diversity mechanisms can be properly customized in order to improve the plethora of search-based approaches that have been recently proposed to support software engineering tasks (e.g., refactoring and scheduling) but that do not properly consider the diversity as an important issue.

Chapter 9

Conclusion and feature work

Contents

9.1 Summary of Contributions 216
9.2 Future Work 218

9.1 Summary of Contributions

This thesis investigated and proposed the usage of search based approaches to reduce the effort of software maintenance and software testing with particular attention to four main activities: (i) program comprehension; (ii) defect prediction; (iii) test data generation and (iv) test suite optimization for regression testing. For program comprehension and defect prediction this thesis provided their first formulations as optimization problems and then proposed the usage of genetic algorithms to solve them (single-objective GAs and MOGAs respectively). Test data generation and test suite optimization have been extensively investigated as search-based problems in literature. However, this thesis presented diversity preserving mechanisms in order to overcome some limitations of evolutionary testing techniques.

The first part of the thesis focused on two semi-automated techniques used to support the software engineer during software maintenance tasks: IR-based program comprehension and defect prediction. In this thesis we noted that these techniques pose the software engineer in front of many possible choices (different instantiations of an IR process, or multiple criteria when using defect prediction models) and the problem of selecting the best choice is not trivial, task and dataset dependent. Thus, we illustrated the benefits of using search-based approaches for calibrating/tunig these techniques.

Chapter 3 investigated a search-based approach to automatically assemble a (near) optimal IR process when solving software engineering problems such as traceability link recovery or feature location. It presented a sound approach —named LSI-GA— to find a (near) optimal configuration of a generic IR process, and a further approach —named LDA-GA— to find (near) optimal calibrations for LDA. The conducted empirical studies involved four different software engineering tasks and demonstrate that: (i) the IR processes instantiated by LSI-GA significantly outperform those assembled according to what previously done in literature; (ii) the performances achieved by LSI-GA are not significantly different from the performances of the global optimum, i.e., the ideal IR process that can be combinatorially built by considering all possible combinations of treatments for the various phases of the IR process, and by having a labeled training set available (i.e., by using a supervised approach); (iii) applying LDA to software engineering tasks requires a careful calibration due to its high sensitivity to different parameter settings; (iv) LDA-GA is able to identify LDA configurations that lead to higher accuracy as compared to alternative heuristics; (v) the performance of LDA-GA are comparable to the best results obtained from the combinatorial search and by having a labeled training set available (i.e., by using a supervised approach). It is also important to highlight that both LDA-GA and LSI-GA are unsupervised since they estimated the *goodness* of an IR process or of a LDA configuration using quality of clustering metrics. Thus, the proposed approaches are unsupervised and task independent.

Chapter 4 formulated the problem of finding good cross-project defect prediction models as a multi-objective problem where multiple and contrasting goals have to be taken

into account. The Chapter presented MODEP, a search-based techniques based on MOGAs, which produces a Pareto front of predictors that allow to achieve different trade-offs between the cost of code inspection and the amount of defect-prone classes correctly classified. An empirical study conducted on 10 open-source software projects demonstrated that: (i) MODEP can be applied to any traditional machine learning technique (for instance in Chapter 4 we used logistic regression or a decision tree) and considering different software criteria, such as the inspection cost, the number of defect-prone classes tested, and the number of defects that can be discovered by the analysis/testing; (ii) MODEP allows to achieve a better cost-effectiveness than single-objective predictors trained with both a within or cross-project strategy; (iii) MODEP outperforms a state-of-the-art approach for cross-project defect prediction [86], based on local prediction among classes having similar characteristics. Specifically, MODEP achieves, at the same level of cost, a significantly higher recall (based on both the number of defect-prone classes and the number of defects). Finally, while traditional defect prediction approaches provides only one prediction model, MODEP allows to achieve different trade-offs between the cost of code inspection (measured in terms of KLOC of the source code artifacts) and the amount of defect-prone classes or number of defects that the model can predict (i.e., recall). In this way, for a given budget (i.e., LOCH that can be reviewed or tested with the available time/resources) the software engineer can choose a predictor that (a) maximizes the number of defect-prone classes tested (which might be useful if one wants to ensure that an adequate proportion of defect-prone classes has been tested), or (b) maximizes the number of defects that can be discovered by the analysis/testing.

The second part of the thesis focused on evolutionary software testing with particular attention to test data generation and test suite optimization. Along with their strengths, testing techniques based on evolutionary algorithms have a set of challenges and open issues. We highlight that one of these issues is the genetic drift, or loss of diversity. For these reasons, we proposed a novel diversity preserving techniques based on Singular Value Decomposition (SVD) [322], to estimate the evolution directions of a population across different generations to promote the exploration of unexplored regions by creating new individuals with orthogonal evolution directions.

Chapters 6 provided a preliminary analysis of the benefits of the proposed techniques when integrated within both GAs and MOGAs (obtaining the SVD augmented algorithms named SVD-GA and SVD-NSGA-II respectively). Specifically, the proposed techniques have been evaluated on 15 single-objective numerical test problems and 12 multi-objective numerical test problems with varying problem dimensions. The results achieved on two empirical studies indicate the superiority of the proposed algorithms if compared to their original versions GAs and NSGA-II. After this preliminary analysis on numerical problems, the SVD-based technique have been evaluated in the context of software testing.

Chapter 7 presented three improved variants of the basic SVD-GA, which are the history aware SVD-GA, reactive GA, and combined history aware and reactive SVD-GA in the context of evolutionary test data generation. Experimental studies on 17 Java classes extracted from widely used open source libraries showed that: (i) the proposed diversification schemata improve over basic GA in terms of effectiveness (i.e., branch coverage achieved) in 8 of the tested classes; (ii) the proposed diversification schemata significantly improved efficiency in 6 out of 7 tested classes where the effectiveness was not significantly improved.

Chapter 8 presented an improved variant of the basic SVD-NSGA-II, named DIV-GA, for solving multi-objective test suite optimization problem. Such a variant combines the basic SVD-NSGA-II algorithm with the orthogonal design in order to both (i) generate a well diversified initial population and (ii) maintain diversity during the evolution of MOGAs. The empirical study conducted on 11 open source programs and relative test suites shows that DIV-GA outperforms both additional greedy algorithm and NSGA-II, which were considered as the best optimizers for multi-objective test case selection problem [68, 27, 65, 34, 35, 36]. In particular, DIV-GA allows (i) to generate more optimal trade-offs with respect to the other optimizers when considering two and three test case selection criteria; (ii) to select sub-test suites that are more cost-effective.

9.2 Future Work

There is a large set of possible improvements that can be studied in future. For example, future work will be devoted to further experiment and assess the proposed search-based approaches in larger software projects for all the experimented software engineering problems. For what concern the program comprehension, we plan to use a more sophisticated evolutionary algorithm, employing genetic programming (GP) to assemble different phases in different ways, including creating ad-hoc weighting schemata [212] or extracting ad-hoc elements from artifacts to be indexed, e.g., only specific source code elements, only certain parts-of-speech. Also, we plan to also optimize the choice of the most appropriate IR algebraic method, also in cases for which such a method does not imply document clustering.

Concerning defect prediction problems, future work aims at considering different kinds of cost-effectiveness models. As said, we considered LOCH is a proxy for code inspection cost, but certainly is not a perfect indicator of the cost of analysis and testing. Alternative cost models, better reflecting the cost of some testing strategies (e.g., code cyclomatic complexity for white box testing, or input characteristics for black box testing) must be considered. Last, but not least, we plan to investigate whether the proposed approach could be used in combination with—rather than as an alternative to—the local prediction approach [86].

For software testing, we plan to support SVD-based diversity mechanisms with non-numeric data types (e.g. strings) as well as an arbitrary number of method call sequences as part of our future extension. For such an extension, the non-numeric types and the method call sequences need to be mapped to a vector space in which a meaningful notion of evolution

direction can be defined. An idea to proceed along these directions involves mapping a variable size sequence (of method calls) to fixed size *feature vectors*, which represent the frequency of occurrence of each feature in an individual. Similarly, feature vectors can be used to represent strings of arbitrary length by extracting pairwise Longest Common Substrings (LCS) from the strings into a feature set and encoding each string as a feature vector containing each feature (LCS). SVD will then be applied to the feature vectors, so as to explore orthogonal regions in the space of feature vectors. In a similar vein, since the current approach considers vectors of fixed size, we intend to investigate an extension of our technique that is able to address vectors of variable dimensions by using complex number for representing variable length candidate solutions. We plan to integrate all these future extensions and enhancements of our implementation into the publicly available tool EvoSuite.

Finally, concerning the test suite optimization problems, we plan to verify whether the proposed diversity mechanisms can be properly customized in order to improve the plethora of search-based approaches that have been recently proposed to support software engineering tasks (e.g., refactoring and scheduling) but that do not properly consider the diversity as an important issue.

Appendix A

SVD-GA: Performances of the Experimented Algorithms

In this appendix we present the results achieved when considering a different problem dimensions with respect to those reported in Chapter 6.

A.1 Empirical study on single-objective GAs

Table A.1 reports the descriptive statistics of the error values achieved by the three experimented optimizers, namely GA, CMA-ES, and SVD-GA when the number of independent variables was set to $n = 100$. When comparing the performances of SVD-GA with GA, we can note that the proposed diversity preserving technique turns out to be very useful for highly dimensional multimodal problems for which the number of local optima is very high. As we can see the results achieved are in-line with those achieved when n=50 reported in Chapter 6.

For the unimodal test functions, Table 6.3 shows —for functions $f_9 - f_{15}$— the descriptive statistics of the error values and the success rate and ERT values (at different tolerance levels), respectively, achieved by the three experimented optimizers (i.e., DC-GA, SVD-NSGA-II, and CMA-ES). As we can see the results achieved are in-line with those achieved when n=100 reported in Chapter 6. As we can see, the SVD-based diversity-preserving technique turns out to be very useful for these unimodal problems too.

A.2 Empirical study on MOGAs

Table A.3 reports mean, median and standard deviation of the IGD metrics achieved using NSGA-II and multi-objective SVD-NSGA-II for the two-objective test problems. Also in this case, the results achieved are in-line with those achieved when n=50. SVD-NSGA-II is able

Table A.1: Function error values achieved on multimodal test functions by GA, SVD-GA, and CMA-ES when n=100. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference.

f	# Fun. Eval.	DC-GA			SVD-GA			CMA-ES		
		Median	Mean	St. Dev.	Median	Mean	St. Dev.	Median	Mean	St. Dev.
f1	10^3	8.41e+2	8.41e+2	4.22e+1	**4.95**	**67.56**	1.12e+2	2.44e+3	2.47e+3	2.12e+2
	10^4	3.75e+2	3.62e+2	3.79e+1	**0**	**8.61**	22.42	8.74e+3	8.86e+2	1.10e+2
	10^5	1.20e+2	1.22e+2	2.01e+1	**0**	**0**	0	1.15e+2	1.33e+2	4.37e+1
	10^6	3.82e+1	3.85e+1	5.21	**0**	**0**	0	2.98	9.14	3.07e+2
f2	10^3	1.53e+1	1.54e+1	6.00e+1	**1.92e-1**	**3.23e-1**	3.01e-1	2.16e+1	2.16e+1	4.77e-2
	10^4	1.03e+1	1.06e+1	1.11	**8.61e-14**	**7.12e-13**	1086e-11	2.16e+1	2.16e+1	5.32e-2
	10^5	2.60	2.61	2.93e-1	**4.44e-15**	**5.57e-15**	3.51e-15	2.16e+1	2.16e+1	4.81e-2
	10^6	2.60	2.61	2.95e-1	**4.44e-15**	**5.29e-15**	3.73e-15	2.16e+1	2.10e+1	8.05e-1
f3	10^3	2.99e+2	2.89e+2	4.43+1	**2.43e-1**	**4.71e-1**	4.21e-1	1.92e+2	1.99e+3	5.73e+2
	10^4	4.41e+1	4.46e+1	7.07	**0**	**1.58e-15**	7.22e-15	3.61e-3	3.87e-3	1.98e-3
	10^5	8.55e-1	8.71e-1	8.50e-2	**0**	**0**	0	2.22e-16	2.62e-16	1.66e-16
	10^6	2.42e-3	5.12e-3	4.74e-3	**0**	**0**	0	2.22e-16	2.13e-16	2.15e-16
f4	10^3	5.87e+6	7.97e+6	6.63e+6	**9.61e-1**	**9.77e-1**	1.60e-1	7.37	1.96e+3	6.86e+3
	10^4	4.76e+1	1.90e+2	3.89e+2	**2.71e-2**	**2.80e-2**	1.08e-2	1.37	3.44e+2	1.72e+2
	10^5	3.11	2.88	7.82e-1	**8.64e-13**	**8.93e-13**	4.02e-13	4.47e-1	2.83	3.95
	10^6	2.88e-6	2.49e-3	1.24e-2	**4.71e-33**	**4.71e-33**	6.98e-49	5.63e-16	1.39e-15	1.92e-15
f5	10^3	3.58e+7	4.03e+7	2.53e+7	*13.59*	**13.40**	1.64	2.13	1.41e+6	4.67e+6
	10^4	9.63e+4	1.37e+5	1.54e+5	**2.97**	**2.92**	6.87e-1	8.05e-1	2.47e+4	1.10e+5
	10^5	3.79	4.33	1.81	**4.17e-9**	**4.66e-9**	3.00e-9	1.94e-2	6.72e-2	1.61e-1
	10^6	8.14e-5	1.05e-4	9.13e-5	**1.65e-32**	**1.35e-32**	5.87e-48	6.39e-16	5.18e-12	2.59e-11
f6	10^3	1.51e+3	1.50e+3	6.46e+1	**1.59e+3**	**1.49e+3**	65.57	2.50e+3	2.55e+3	2.59e+2
	10^4	6.50e+2	6.49e+2	4.65e+1	**4.70e+2**	**4.66e+2**	39.67	9.52e+2	9.60e+2	2.19e+2
	10^5	1.55e+2	1.58e+2	1.99e+1	**3.32e+1**	**3.31e+1**	4.22	1.19e+2	2.21e+2	2.96e+2
	10^6	2.80e+1	2.79e+1	5.79	**5.39e-4**	**1.20e-1**	3.30e-1	3.98	1.36e+2	3.67e+2
f7	10^3	2.04e+1	2.04e+1	1.49e-1	2.05e+1	2.05e+1	1.54e-1	2.16e+1	2.16e+1	5.43e+2
	10^4	1.65e+1	1.62e+1	1.04	**1.43e+1**	**1.44e+1**	9.77e-1	2.16e+1	2.16e+1	4.28e-2
	10^5	3.61	3.60	4.09e-1	**1.50**	**1.49**	2.54e-1	2.16e+1	2.16e+1	4.96e-2
	10^6	1.23	1.13	3.84e-1	**9.23e-4**	**9.27e-4**	1.29e-4	2.16e+1	21.08e+1	7.63e-1

to converge better in all the two-objective problems independently. Indeed, median, mean and standard deviation of IGD values achieved by SVD-NSGA-II are always smaller than those of NSGA-II. Moreover, the Wilcoxon rank sum test reveals that such differences are statistically significant.

A.2.1 Graphs of Non-Dominated Solutions obtained by SVD-NSGA-II and NSGA-II

In this appendix, we report the graphs of the non-dominated solutions obtained by SVD-NSGA-II and NSGA-II achieved on all the two-objective and three-objective test functions. Figures A.1, A.2, and A.3 show all non-dominated solutions obtained after $5 \cdot 10^5$ generations with NSGA-II and SVD-NSGA-II on the MOP2, ZDT3, and ZDT6 test problems, respectively, when the space dimension is n=100, while n=50 the graphs are shown in Chapter 6.

Figure A.4 shows all non-dominated solutions obtained after $5 \cdot 10^6$ function evaluation with NSGA-II and SVD-NSGA-II on the MOP1 test problem. Such a problem has a huge search space, while the Pareto optimal front is located in a small search sub-space of the whole search space. This problem is usually used for measuring the speed of an evolutionary algorithm to converge to the Pareto optimal front. The figure clearly demonstrates the

Table A.2: Function error values achieved on unimodal test functions by GA, SVD-GA, and CMA-ES when n=100. Values shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference.

f	Eval.	GA			SVD-GA			CMA-ES		
		Median	Mean	St. Dev.	Median	Mean	St. Dev.	Median	Mean	St. Dev.
f8	10^3	1.92e+3	1.92e+3	1.35e+2	**7.82e-1**	**4.85e-1**	1.24e-1	1.81e+3	2.05e+3	7.50e+2
	10^4	1.84e+2	1.81e+2	3.33e+1	**0**	**1.36e-14**	2.34e-14	4.38e-3	4.73e-3	2.94e-3
	10^5	1.14	1.15	2.72e-2	**0**	**5.68e-15**	1.16e-14	2.84e-14	3.30e-14	1.06e-14
	10^6	1.46e-2	1.52e-2	7.03e-3	**0**	**0**	0	*0*	1.14e-15	5.68e-15
f9	10^3	3.06e+4	3.14e+4	5.75e+3	**3.82**	**2.09e+1**	2.09e+1	2.10e+5	2.27e+5	7.52e+4
	10^4	4.62e+3	4.73e+3	8.84e+2	**8.64e-24**	**1.780e-21**	8.68e-21	2.07e-4	2.63e-4	1.41e-4
	10^5	3.05	3.31	8.62e-1	**1.22e-148**	**1.32e-143**	4.57e-143	4.93e-16	4.63e-16	1.38e-16
	10^6	1.35e-3	1.33e-3	2.43e-4	**0**	**0**	0	4.63e-16	4.60e-16	1.07e-16
f10	10^3	1.33e+2	1.37e+2	1.89e+1	**1.57**	**2.07**	1.47	2.46e+106	2.85e+124	1.07e+124
	10^4	5.48e+1	5.32e+1	6.52	**5.94e-12**	**3.32e-11**	7.35e-11	4.71e+100	1.04e+116	4.03e+117
	10^5	2.05	2.14	3.67e-1	**7.28e-71**	**1.07e-68**	3.08e-68	1.28e+48	4.21e+64	1.63e+65
	10^6	5.12e-2	5.44e-2	1.09e-2	**0**	**0**	0	1.70e-5	5.75	2.22e+1
f11	10^3	1.10e+5	1.19e+5	3.60e+4	61.06e+2	4.18e+3	7.68e+3	**1.33e+1**	**8.08e+1**	1.56e+2
	10^4	3.22e+4	3.19e+4	1.08e+4	1.72	9.18e+2	2.42e+3	**2.34**	**6.63e+1**	1.99e+2
	10^5	4.73e+3	4.72e+3	7.71e+2	**1.89e-9**	**4.76**	2.38e+1	5.76e-3	4.11e-1	9.53e-1
	10^6	1.47e+2	1.43e+2	1.47e+1	**1.24e-28**	**1.05e-16**	3.04e-16	2.92e-16	1.67e-15	3.30e-15
f12	10^3	4.41e+1	4.41e+1	4.16	**2.18e-1**	**1.11-2**	297e-1	4.61e+2	4.54e+2	5.74e+1
	10^4	2.96e+1	3.06e+1	2.96	**4.58e-8**	**1.11-2**	214e-3	2.03e+2	2.07e+2	2.92e+1
	10^5	9.96	1.00e+1	8.60e-1	**1.00e-8**	**1.79-3**	6.82e-3	7.73	8.56	3.48
	10^6	6.77e-1	7.08e-1	1.33e-1	**8.43e-16**	**2.04e-10**	6.92e-10	2.23e-1	2.87e-1	2.65e-1
f13	10^3	3.12e+2	3.13e+2	1.04e+1	**3.09e+2**	**3.07e+2**	1.09e+1	2.72e+4	2.74e+4	6.19e+3
	10^4	1.69e+2	1.69e+2	8.94	**1.33e+2**	**1.32e+2**	8.62	1.97e+2	2.15e+2	1.01e+2
	10^5	5..19e+1	5.24e+1	6.29	**3.74**	**3.92**	2.30	1.81e+2	2.02e+2	9.83e+1
	10^6	2.39	2.35	1.80	**9.66e-13**	**9.39e-13**	5.71e-14	1.79e+2	2.19e+2	1.40e+2
f14	10^3	2.36e+5	2.37e+5	1.85e+4	2.30e+5	2.33e+5	1.79e+4	2.34e+5	2.24e+5	6.05e+4
	10^4	2.25e+4	2.25e+4	3.68e+3	1.25e+4	1.23e+4	2.15e+3	**2.30e-4**	**2.56e-4**	1.43e-4
	10^5	2.26e+1	2.26e+1	5.40	7.50e-1	8.22e-1	2.19e-1	**5.68e-14**	**7.73e-14**	2.78e-14
	10^6	7.83e-3	7.83e-3	1.48e-3	5.48e-5	5.57e-5	1.11e-5	**0**	**0**	0
f15	10^3	1.20e+8	1.25e+8	2.99e+7	7.61e+7	8.02e+7	3.26e+7	**2.88e+1**	**2.13e+2**	6.58e+2
	10^4	1.73e+5	1.80e+5	2.72e+4	1.44e+5	1.36e+5	3.11e+4	**8.35**	**1.11e+2**	3.79e+2
	10^5	3.34e+4	3.27e+4	4.53e+3	1.28e+4	1.31e+4	2.40e+3	**1.62e-3**	**4.49e-1**	1.79
	10^6	2.65e+3	2.68e+3	4.88e+2	1.16e+2	1.13e+2	24.02	**0**	**0**	0

abilities of SVD-NSGA-II in converging to the true front, while NSGA-II is quite far from the Pareto-optimal region.

Figure A.5 shows the Pareto fronts obtained by the two algorithms on the shifted SDT1 test problem. SVD-NSGA-II performed better than NSGA-II in terms of both convergence and distribution of solutions. Indeed, NSGA-II was not able to maintain enough diversified non-dominated solutions in the final population and the distribution of the solutions is worst if compared to that obtained in the final population of SVD-NSGA-II. Moreover, SVD-NSGA-II achieved a front that is more close to the optimal front.

Figure A.6 shows the results achieved on the shifted ZDT2 test problem. This problem has a non-convex Pareto-optimal front. Although the convergence is not a difficulty here with both of the algorithms, SVD-NSGA-II showed better convergence to the optimal front and it has found a better Pareto front than NSGA-II.

The problem ZDT4 (see Figure A.7) has an exponential number of different local Pareto-optimal fronts in the search space, and only one among them corresponds to the global Pareto-

Table A.3: IGD values achieved by SVD-NSGA-II and NSGA-II for n=100. Values are shown in bold face for comparisons where the Wilcoxon Rank Sum test indicates a statistically significant difference.

f	Func. Eval.	NSGA-II			SVD-NSGA-II		
		Median	Mean	St. Dev.	Median	Mean	St. Dev
MOP1	$4 \cdot 10^5$	1.05e+6	1.56e+6	1.71e+6	**2.01e-1**	**2.25e-1**	1.02e-1
	$6 \cdot 10^5$	1.05e+6	1.56e+6	1.71e+6	**1.92e-1**	**1.92e-1**	7.53e-3
	$8 \cdot 10^5$	1.05e+6	1.56e+6	1.71e+6	**1.91e-1**	**1.92e-1**	9.36e-3
	10^6	1.05e+6	1.56e+6	1.71e+6	**1.90e-1**	**1.92e-1**	1.43e-2
MOP2	$2 \cdot 10^4$	5.63e-1	5.55e-1	1.43e-1	**5.23e-2**	**5.36e-2**	3.67e-3
	$6 \cdot 10^5$	5.06e-1	4.71e-1	1.68e-1	**4.60e-2**	**4.66e-2**	1.87e-3
	$8 \cdot 10^5$	4.39e-1	4.04e-1	1.69e-1	**4.60e-2**	**4.59e-2**	1.50e-3
	10^6	3.61e-1	3.44e-1	1.69e-1	**4.53e-2**	**4.47e-2**	1.91e-3
Shifted ZDT1	$2 \cdot 10^4$	1.81e-1	2.03e-1	8.11e-2	**1.61e-2**	**1.69e-2**	2.81e-3
	$6 \cdot 10^5$	1.11e-1	1.13e-1	4.99e-2	**8.70e-3**	**8.85e-3**	6.63e-4
	$8 \cdot 10^5$	7.40e-2	7.40e-2	2.93e-2	**7.10e-3**	**7.07e-3**	3.29e-4
	10^6	5.55e-2	5.81e-2	2.53-2	**5.10e-3**	**5.19e-3**	7.777e-4
Shifted ZDT2	$2 \cdot 10^4$	4.74e-1	5.07e-1	2.13e-1	**2.30e-2**	**2.31e-2**	3.32e-3
	$6 \cdot 10^5$	3.20e-1	3.72e-1	2.20e-1	**1.101e-2**	**1.02e-2**	7.66e-4
	$8 \cdot 10^5$	2.87e-1	3.34e-1	2.33e-1	**7.77e-3**	**7.86e-3**	4.71e-4
	10^6	2.72e-1	3.17e-1	2.42e-1	**5.12e-3**	**5.36e-3**	8.81e-4
ZDT3	$2 \cdot 10^4$	1.41e-1	1.42e-1	1.60e-2	**6.48e-3**	**5.74e-2**	1.11e-1
	$6 \cdot 10^5$	1.16e-1	1.17e-1	1.68e-2	**6.30e-3**	**5.73e-2**	1.11e-1
	$8 \cdot 10^5$	1.05e-1	1.06e-1	1.74e-2	**6.31e-3**	**5.72e-2**	1.11e-1
	10^6	9.95e-2	1.00e-1	1.78e-2	**3.45e-3**	**5.53e-2**	1.12e-1
Shifted ZDT4	$2 \cdot 10^4$	3.35e+2	3.28e+2	3.20e+1	**6.06e+1**	**6.26e+1**	1.78e+1
	$6 \cdot 10^5$	2.31e+2	2.17e+2	4.27e+1	**2.02e+1**	**2.04e+1**	4.41
	$8 \cdot 10^5$	1.58e+2	1.55e+2	3.54e+1	**8.73**	**9.14**	2.25
	10^6	1.28e+2	1.28e+2	2.86e+1	**4.40**	**4.46**	1.42
Shifted ZDT6	$2 \cdot 10^4$	1.07	8.30e-1	4.93e-1	**9.49e-2**	**1.05e-1**	3.55e-2
	$6 \cdot 10^5$	1.04	7.48e-1	5.45e-1	**1.42e-2**	**1.54e-2**	3.82e-3
	$8 \cdot 10^5$	1.04	7.33e-1	5.61e-1	**7.21e-3**	**7.60e-3**	1.17e-3
	10^6	1.04	7.32e-1	5.62e-1	**4.88e-3**	**5.02e-3**	6.18e-4

optimal front. On this problem, both NSGA-II and SVD-NSGA-II get stuck at different local Pareto-optimal sets, but the convergence and the ability to find a diverse set of solutions are definitely better with SVD-NSGA-II.

Figure A.8 shows the obtained set of non-dominated solutions after $5 \cdot 10^5$ function evaluations using NSGA-II and SVD-NSGA-II for DTLZ1 (with $n = 20$). The difficulty in this problem is to converge to the Pareto optimal plan. The search space contains $\left(11^8 - 1\right)$ sub-optimal fronts, each of which can attract the experimented algorithm. The figure shows that NSGA-II is able to uniformly maintain solutions in decision search space. However, the obtained solutions are really far from the optimal front which is really smaller (it is the black point near the origin of axis). Instead, SVD-NSGA-II is able to obtain much better and uniform solutions. Similar analysis can be done for $n = 100$ as showed in Figure A.9.

The same considerations hold for all the other three-objective functions, as showed by Figures A.10, A.11, A.12, A.13, A.14, A.15, and A.16. For them SVD-NSGA-II achieved a set of non-dominated solutions that is close to the optimal ones. Instead, even if the set

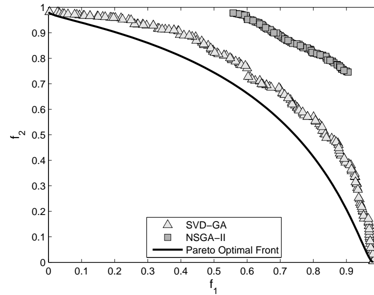


Figure A.1: NSGA-II vs. SVD-NSGA-II on MOP2 test problem when $n = 100$ ($n = 50$ is reported in Chapter 6).

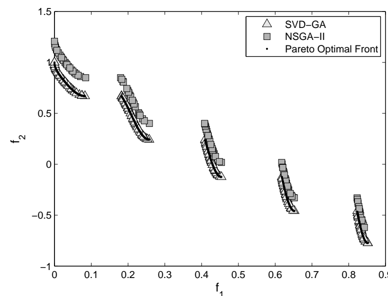


Figure A.2: NSGA-II vs SVD-NSGA-II on ZDT3 test problem when $n = 100$ ($n = 50$ is reported in Chapter 6).

of non-dominated solutions reached by NSGA-II is well distributed, it is very far from the Pareto optimal front.

Particularly interesting is the DTLZ6 test problem. It has four disconnected Pareto-optimal regions in the search space and it was designed for testing the ability to maintain sub-population in different Pareto-optimal regions. For such a problem, SVD-NSGA-II turned out to be the best one from convergence and spread of solutions. Indeed, SVD-NSGA-II reached a set of solutions close to the Pareto optimal front and such a solution are well distributed in all the four disconnected Pareto-optimal regions. Instead, NSGA-II provided solutions that are far from the optimal front and cover only two out of four disconnected Pareto-optimal region.

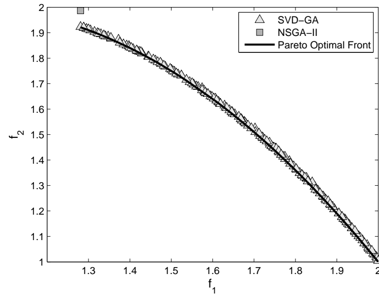
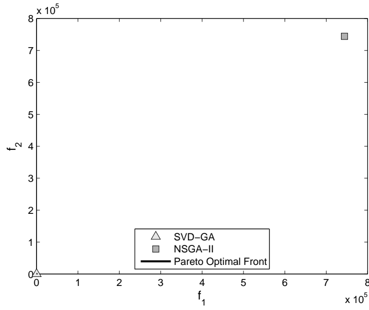
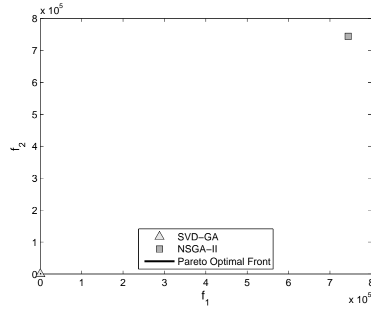


Figure A.3: NSGA-II vs SVD-NSGA-II on ZDT6 test problem when $n = 100$ ($n = 50$ is reported in Chapter 6).

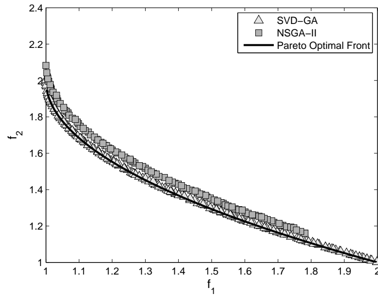


(a) $n = 50$

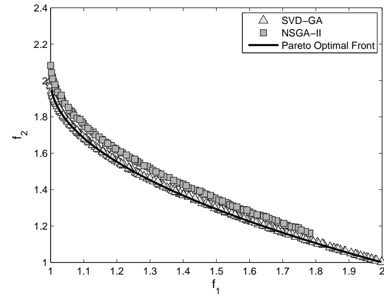


(b) $n = 100$

Figure A.4: NSGA-II vs. SVD-NSGA-II on MOP1 problem.

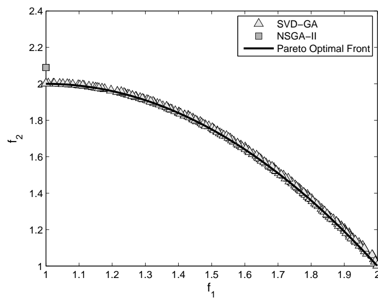


(a) $n = 50$

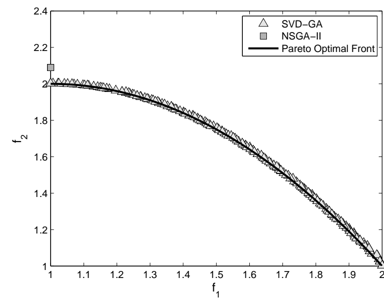


(b) $n = 100$

Figure A.5: NSGA-II vs SVD-NSGA-II on shifted ZDT1 test problem.

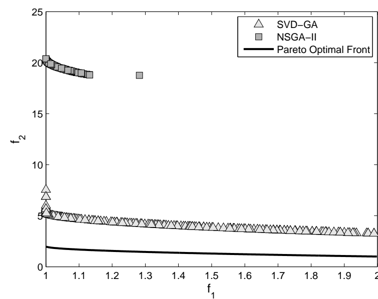


(a) $n = 50$

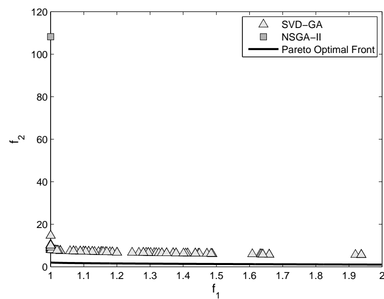


(b) $n = 100$

Figure A.6: NSGA-II vs. SVD-NSGA-II on shifted ZDT2 test problem.

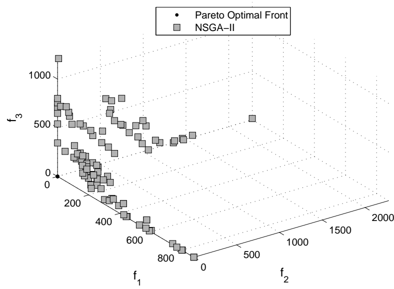


(a) $n = 50$

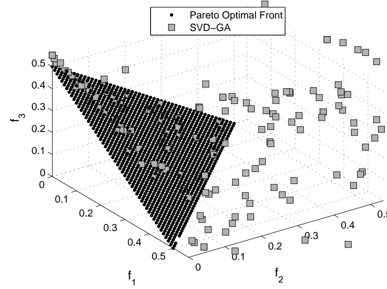


(b) $n = 100$

Figure A.7: NSGA-II vs SVD-NSGA-II on shifted ZDT4 test problem.

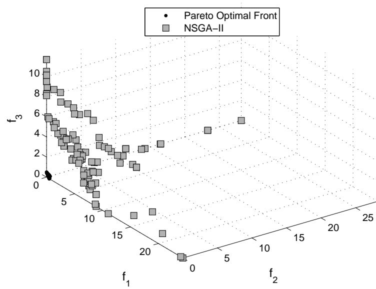


(a) NSGA-II

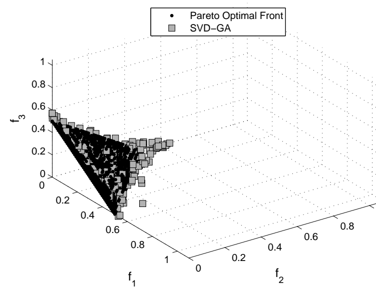


(b) SVD-NSGA-II

Figure A.8: NSGA-II vs. SVD-NSGA-II on DTLZ1 when $n = 50$.

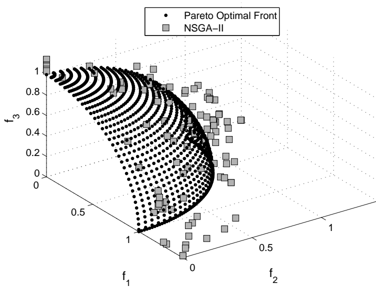


(a) NSGA-II

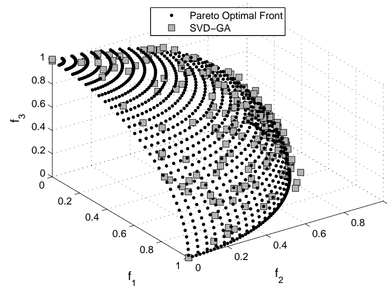


(b) SVD-NSGA-II

Figure A.9: NSGA-II vs. SVD-NSGA-II on DTLZ1 when $n = 100$.

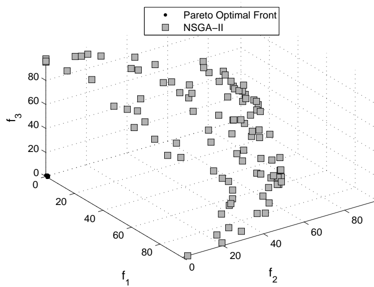


(a) NSGA-II

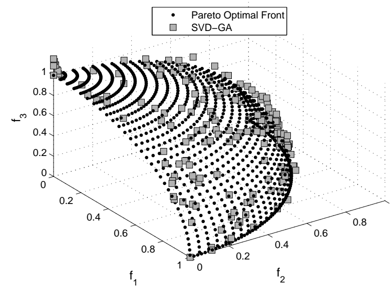


(b) SVD-NSGA-II

Figure A.10: NSGA-II vs. SVD-NSGA-II on DTLZ2 when $n = 50$.

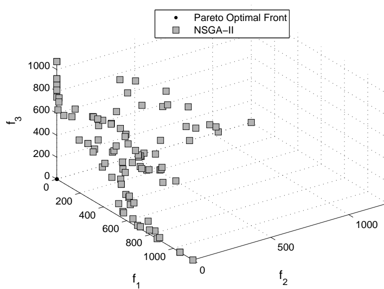


(a) NSGA-II

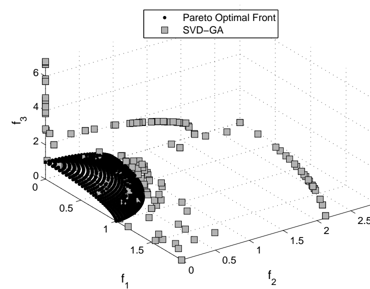


(b) SVD-NSGA-II

Figure A.11: NSGA-II vs. SVD-NSGA-II on DTLZ2 when $n = 100$.

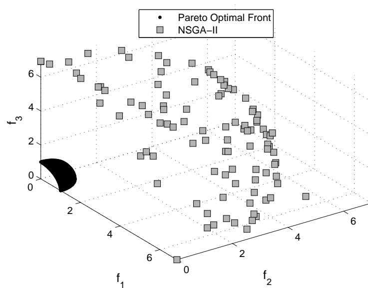


(a) NSGA-II

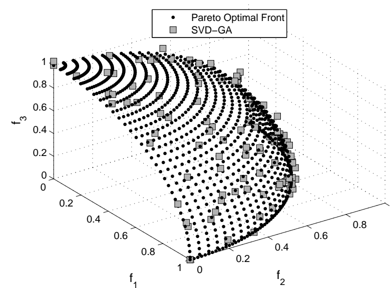


(b) SVD-NSGA-II

Figure A.12: NSGA-II vs. SVD-NSGA-II on DTLZ3 when $n = 10$.

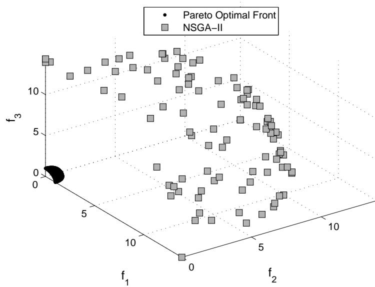


(a) NSGA-II

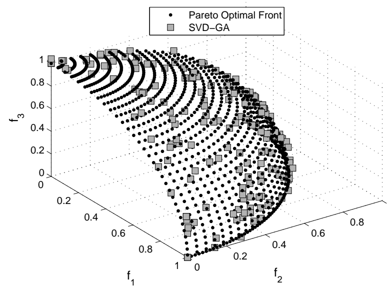


(b) SVD-NSGA-II

Figure A.13: NSGA-II vs. SVD-NSGA-II on DTLZ4 when $n = 50$.

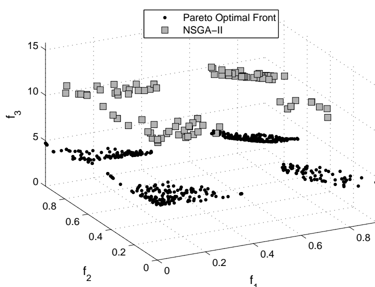


(a) NSGA-II

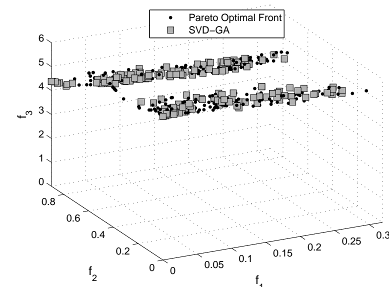


(b) SVD-NSGA-II

Figure A.14: NSGA-II vs. SVD-NSGA-II on DTLZ4 when $n = 100$.

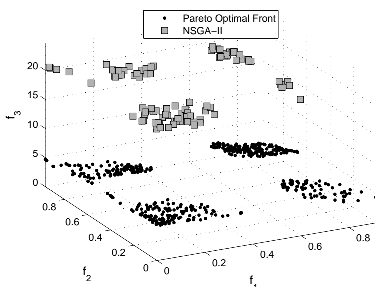


(a) NSGA-II

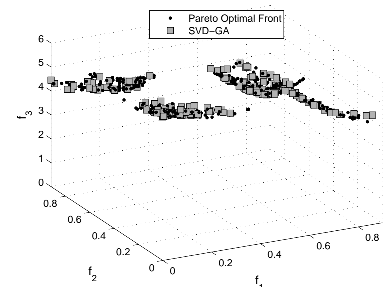


(b) SVD-NSGA-II

Figure A.15: NSGA-II vs. SVD-NSGA-II on DTLZ6 when $n = 50$.



(a) NSGA-II



(b) SVD-NSGA-II

Figure A.16: NSGA-II vs. SVD-NSGA-II on DTLZ6 when $n = 100$.

Appendix B

Binary Representation of Testing Criteria

In this appendix we report the mathematical formulation of the test criteria using a binary vector representation that was successfully used in previous works to deal with the multi-objective test case selection and test suite minimization problems [27, 65, 68]. A solution to the multi-objective test suite optimization problem can be represented as a binary vector X of length n, where the generic element x_i is 1 if test case t_i is selected, 0 otherwise. Starting from this binary representation of a solution, the testing criteria can be easily defined using linear algebra operations.

Statement coverage criterion. The statement level coverage information used in this paper was measured using the GNU compiler, `gcc`, and its code coverage tool, `gcov`. The achievement of a *coverage criterion* can be defined as the product matrix between a solution X multiplied by a matrix that represents the statement level coverage data of the entire test suite [65]. Let m be the number of code statements to be covered, and n the number of test cases in the test suite. We can define a binary matrix A containing the trace data that captures the code statements achieved by each test case. A generic entry $a_{i,j}$ of A is equal to 1 if the ith statement was covered by the jth test case, 0 otherwise [68, 65] according to profiling data provided by `gcov`. Then, the coverage (cov) achieved by the solution represented by X can be measured as follows [68, 65]:

$$cov(X) = \frac{1}{m} \sum_{i=1}^{m} \phi_i \quad \text{where} \quad \phi_i = \begin{cases} 1 & \text{if } g_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{B.1}$$

where $g_i \in G$ denotes the number of times the ith test goal was covered by the selected test cases in X. The vector G can be computed as follows:

$$G = A \cdot X = \begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \tag{B.2}$$

Execution cost criterion. As for the execution cost, in principle we could just measure the test case execution time. However, performing such a measure is not an easy task, because the measure depends on several external factors such as different hardware, application software, operating system, etc. We address this issue by counting the number of executed elementary instructions in the code, instead of measuring the actual execution time. This is consistent with what was done in previous work on multi-objective test case selection [27, 68]. We use the `gcov` tool to measure the execution frequency of every basic block (elementary instruction) composing each statements. When encountering a function call, `gcov` counts the instructions actually executed when invoking the function; also, complex statements may count as multiple elementary instructions (e.g., the `for` statement counts as three instructions, i.e., the initialization, the condition, and the increment). Starting from these execution frequencies, the *execution cost criterion* is defined as follows:

$$cost(X) = \sum_{i=1}^{n} x_i \cdot cost(t_i) \tag{B.3}$$

where $cost(t_i)$ represents the execution frequency of the ith test case.

Past fault coverage criterion. For the third test criteria used in this paper, each program has several faulty versions available from the SIR dataset [343]. SIR provides also information about which test cases are able to reveal these faults. Such information can be used to assign a past fault coverage value to each test case subset, computed as the number of known past faults that this subset is able to reveal in the previous version. Finally, once the past faults coverage data is stored in a binary matrix, the *past fault coverage* criterion can be defined, using again a matrix multiplication. Let F be a binary matrix containing the trace data that captures the past faults covered by each test case. A generic entry $f_{i,j}$ of F is equal to 1 if the ith past fault was covered by the jth test case, 0 otherwise. Then, the past fault coverage (*fault*) achieved by a solution X can be measured as follows:

$$fault(X) = \frac{1}{m} \sum_{i=1}^{m} \varphi_i \text{ where } \varphi_i = \begin{cases} 1 & \text{if } f_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{B.4}$$

where $f_i \in \mathcal{F}$ denotes the number of times the ith fault was covered by the selected test cases in X. The vector $\mathcal{F}$ can be computed as follows:

$$\mathcal{F} = F \cdot X = \begin{bmatrix} f_{1,1} & \dots & f_{1,n} \\ \vdots & \ddots & \vdots \\ f_{k,1} & \dots & f_{k,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \tag{B.5}$$

Using the three test case selection criteria described above, we examine two and three-objectives formulations of the test case selection problem. In particular, we consider the two-objective problem taking into account code coverage and execution cost as contrasting goals, similarly to what done in previous work [27, 68, 11]. Formally, the two-objective test case selection problem can be defined as follows:

Problem 8. Two-objective Test Case Selection Problem: *finding a set of optimal solutions X which maximizes the code coverage and minimizes the execution cost:*

$$\max cov(X) = \frac{1}{m} \sum_{i=1}^{m} \phi_i$$
$$\min cost(X) = \sum_{i=1}^{n} x_i \cdot cost(t_i)$$

For the three-objective formulation, we add past fault detection history as a further objective, as also done in previous work [97, 68, 27]. The three-objective test case selection problem can be defined as follows:

Problem 9. Three-objective Test Case Selection Problem: *finding a set of optimal solutions X which maximizes the code coverage, maximizes the past fault coverage and minimizes the execution cost:*

$$\max cov(X) = \frac{1}{m} \sum_{i=1}^{m} \phi_i$$
$$\min cost(X) = \sum_{i=1}^{n} x_i \cdot cost(t_i)$$
$$\max fault(X) = \frac{1}{h} \sum_{i=1}^{h} \varphi_i$$

Note that, besides the test case selection criteria defined above, it is possible to formulate other criteria, e.g., based on data-flow coverage or even functional requirements just providing a clear mapping between tests and criterion-based requirements. However, it is important to highlight that the goal of this work is not to analyse which set of test criteria is the most effective for regression testing. The formulations are used to illustrate how the proposed diversity-preserving technique can be applied to any number and kind of testing criteria to be satisfied.

Appendix C

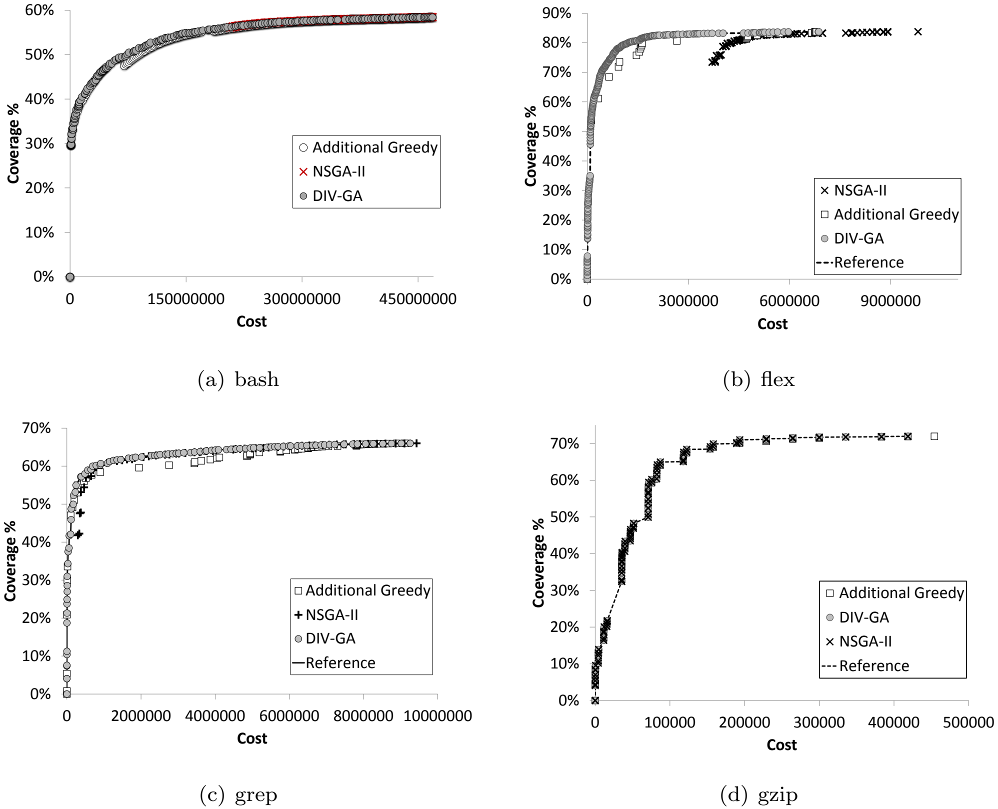
Multi-Objective Test Suite Optimization

In this appendix, we report the results achieved in our empirical study. Specifically, we report all the Pareto frontiers achieved when comparing the proposed DIV-GA with the additional greedy algorithm and the island version of NSGA-II.

C.1 RQ_1: To what extent does DIV-GA produce near optimal solutions, compared to alternative test case selection techniques?

Figures C.1-C.3 provide a graphical comparison between the Pareto frontiers obtained by the three algorithms and a "reference" Pareto frontier, built as explained in section 4. As it can be noticed, the Pareto frontiers provided by DIV-GA are much closer to the reference Pareto frontiers (often the two frontiers are perfectly overlapped) than the Pareto frontiers provided by NSGA-II and the additional greedy. The additional greedy algorithm provides solutions that are, in some cases, quite close to the reference frontiers. However, the majority of them are dominated by the solutions produced by DIV-GA. Instead, NSGA-II produces solutions quite far from the optimal set of solutions (e.g., on `printtokens`). Only on `gzip` all the algorithms turned out to be close to the reference frontier. Particularly interesting are the results achieved for all the programs in the Siemens suite —i.e. `printtokens`, `printtokens2`, `schedule`, and `schedule2`— where NSGA-II is very far from the optimal Pareto frontier while DIV-GA provides (near) optimal frontiers. Finally, for what concerns the uniformity of the distribution of the solutions over the produced Pareto frontiers, we can also observe that DIV-GA also provides a wider diversity of non-dominated solutions with higher coverage and uniformity along the Pareto frontier than the other algorithms. Conversely, NSGA-II and the additional greedy algorithm provide solutions which are not well distributed uniformly

along the Pareto frontiers, i.e., more solutions for higher coverage levels and leaving the rest of the Pareto frontiers quite unexplored.

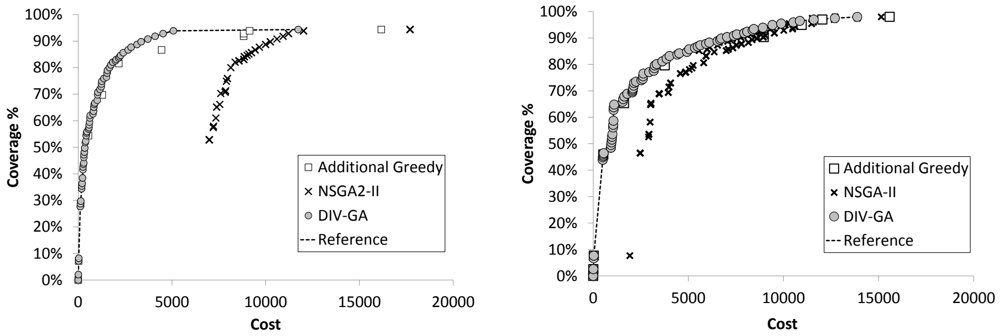


(a) bash (b) flex

(c) grep (d) gzip

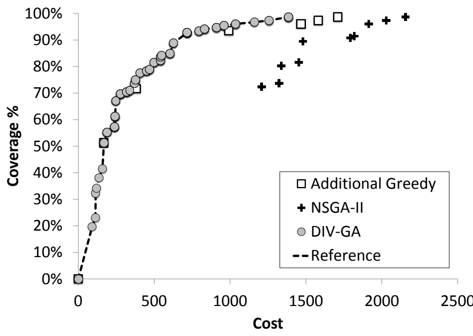
Figure C.1: Pareto frontiers achieved on *bash*, *flex*, *grep* and *gzip* for two-objectives test suite optimization problem

Figures C.4-C.14 show the results for the three-objective formulation. The 3D plots displays the solutions produced by (i) DIV-GA, (ii) the additional greedy algorithm, (iii) NSGA-II, and (iv) the reference Pareto frontier (denoted using black dots). The additional greedy algorithm produces solutions that are quite close to the reference frontier. However, the number of the produced solutions is really small if compared to the reference frontier. DIV-GA always produces three-objective solutions stated in the reference frontier: in all cases the Pareto frontier of DIV-GA exactly overlaps the reference frontier. Hence, DIV-GA always produces solutions that are non-dominated by any other algorithm. Instead, NSGA-II produces (near) optimal solutions only in a few cases. For example, on `gzip` the Pareto frontier obtained by NSGA-II is quite close to the reference Pareto frontier (see Figure C.7), while on `printtokens`, the solutions obtained by NSGA-II are quite far from the reference Pareto frontier (see Figure C.8).

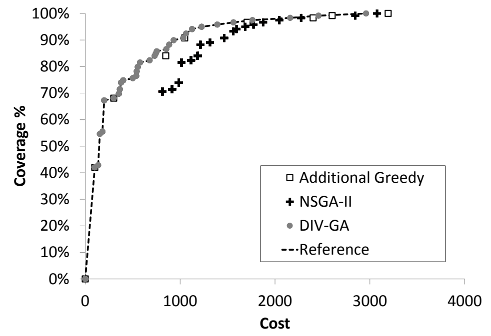


(a) printtokens

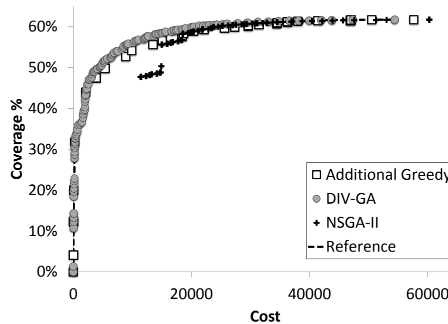
(b) printtokens2



(c) schedule

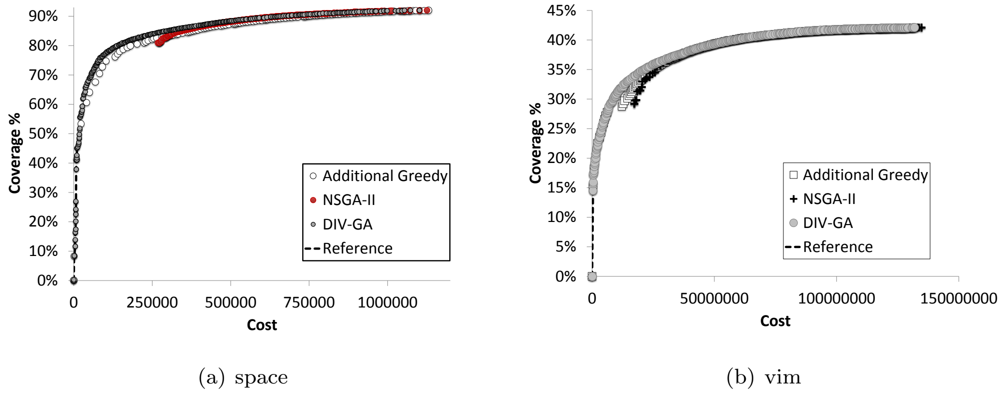


(d) schedule2



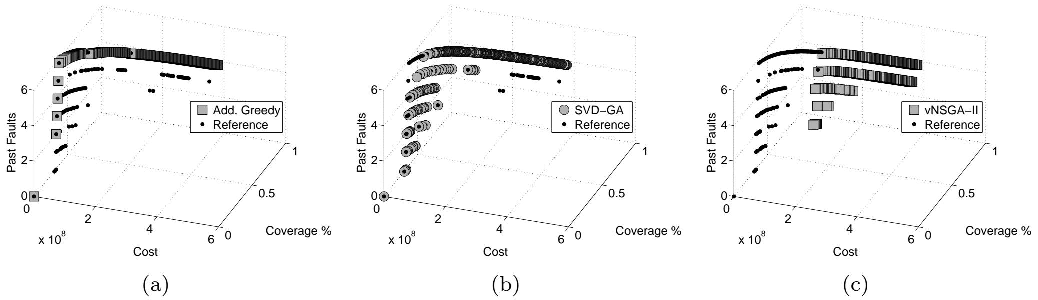
(e) sed

Figure C.2: Pareto frontiers achieved on *printtokens*,*printtokens2*, *schedule*, *schedule* and *sed* for two-objectives test suite optimization problem



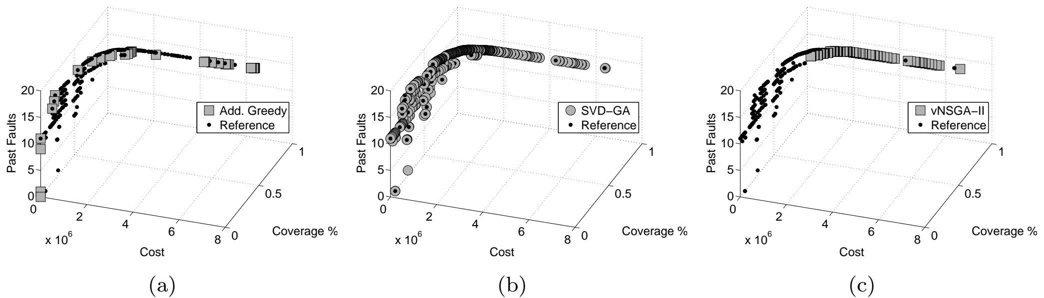
(a) space (b) vim

Figure C.3: Pareto frontiers achieved on *space* and *vim* for two-objectives test suite optimization problem



(a) (b) (c)

Figure C.4: Three-objective Pareto Frontiers achieved on *bash*



(a) (b) (c)

Figure C.5: Three-objective Pareto Frontiers achieved on *flex*.

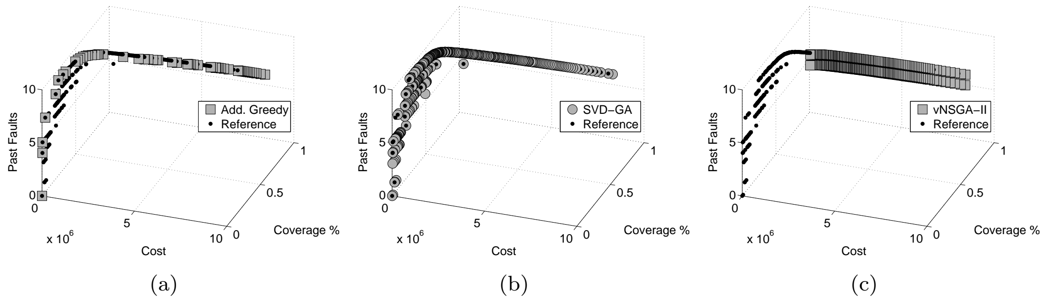


Figure C.6: Three-objective Pareto Frontiers achieved on *grep*

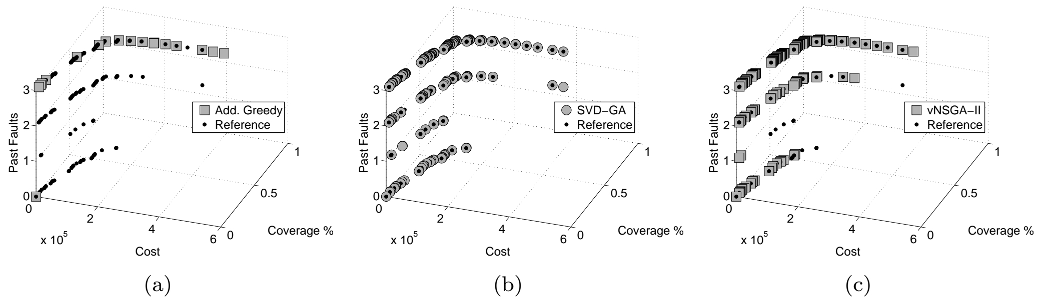


Figure C.7: Three-objective Pareto Frontiers achieved on *gzip*.

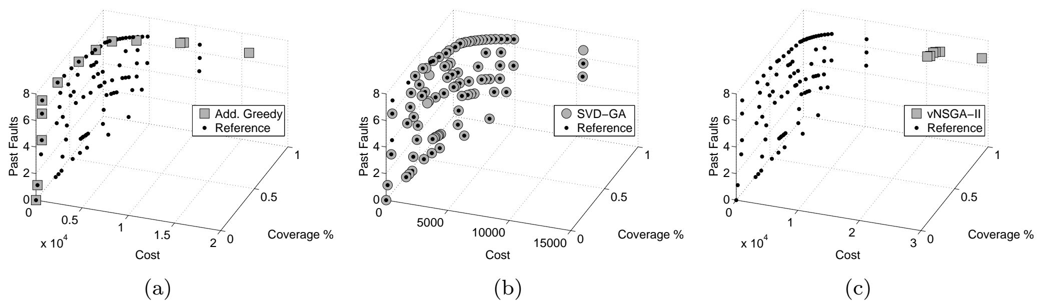
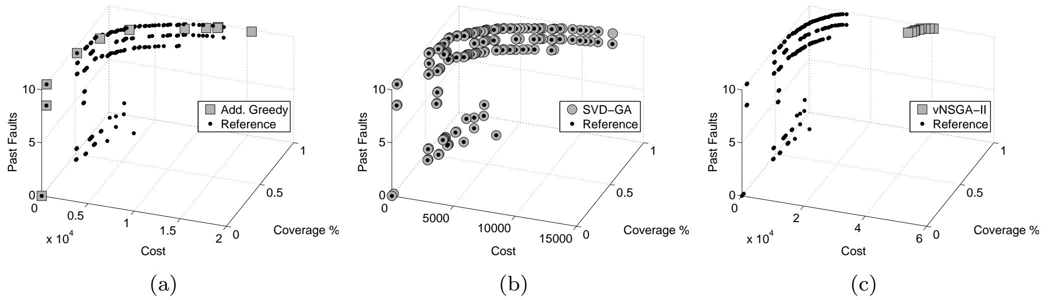
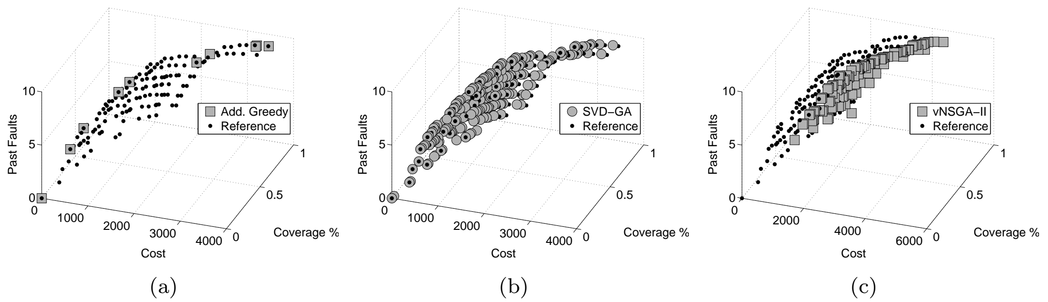


Figure C.8: Three-objective Pareto Frontiers achieved on *printtokens*.



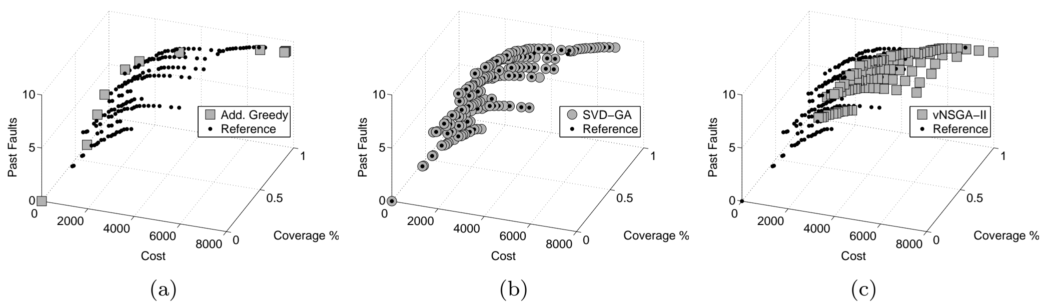
(a) (b) (c)

Figure C.9: Three-objective Pareto Frontiers achieved on *printtokens2*



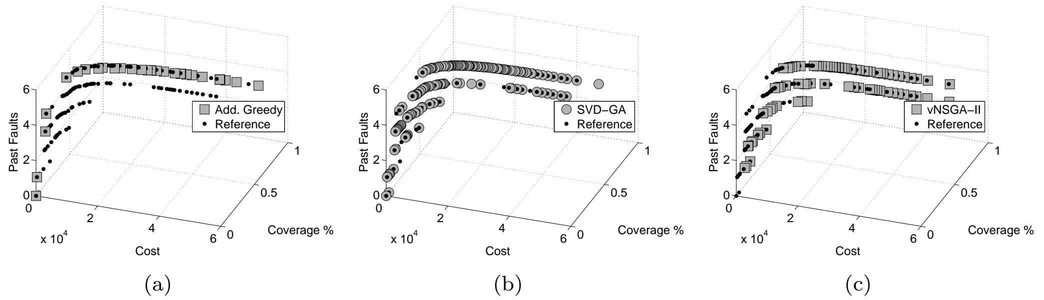
(a) (b) (c)

Figure C.10: Three-objective Pareto Frontiers achieved on *schedule*



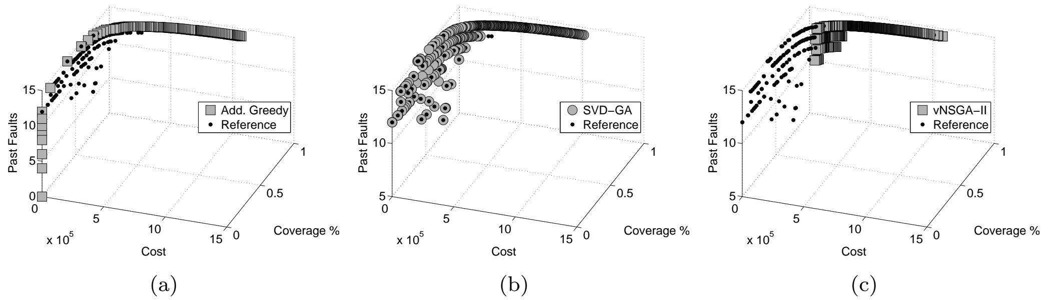
(a) (b) (c)

Figure C.11: Three-objective Pareto Frontiers achieved on *schedule2*



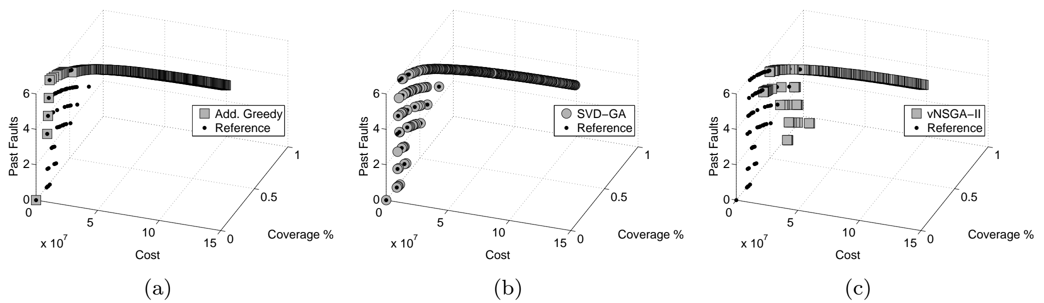
(a) (b) (c)

Figure C.12: Three-objective Pareto Frontiers achieved on *sed*



(a) (b) (c)

Figure C.13: Three-objective Pareto Frontiers achieved on *space*

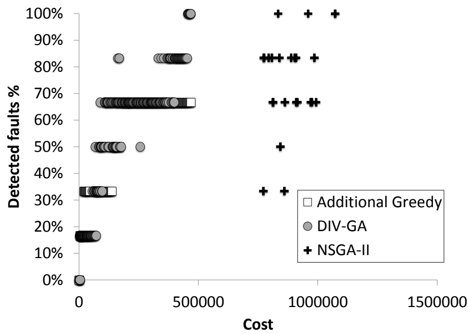


(a) (b) (c)

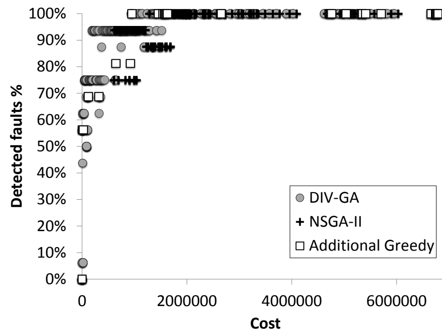
Figure C.14: Three-objective Pareto Frontiers achieved on *vim*

C.2 RQ_2: What is the cost-effectiveness of DIV-GA, compared to alternative test case selection techniques?

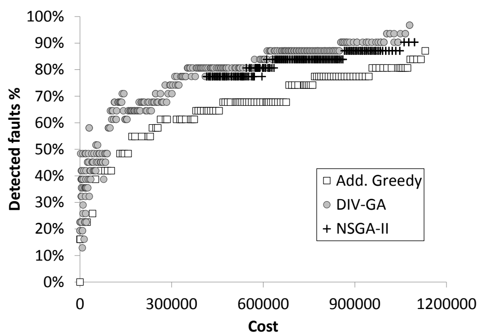
Figures C.15-C.17 plot the percentage of faults detected by the solutions (sub-test suites) provided by the three algorithms at same level of execution cost. The goal of such an analysis is to provide a graphical comparison of the percentage of faults detected by the different solutions (sub-test suites) at same level of execution cost. We can observe that the sub-test suites selected by DIV-GA are able to detect more faults than the additional greedy with lower execution cost. For example, on `space` the test suites optimized by DIV-GA can detect 100% of faults, while the percentage of faults produced by the other two algorithms is lower at the same level of execution cost.



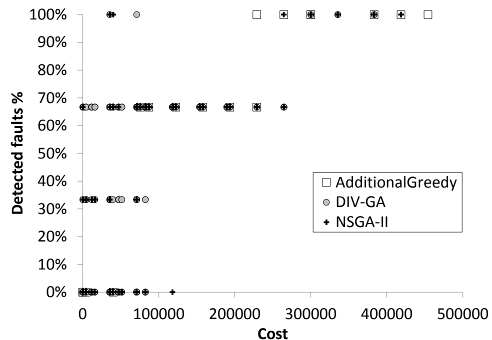
(a) bash



(b) flex



(c) space



(d) gzip

Figure C.15: Effectiveness of the achieved sub-test suites for two-objective test case selection on *bash*, *flex*, *grep* and *gzip*.

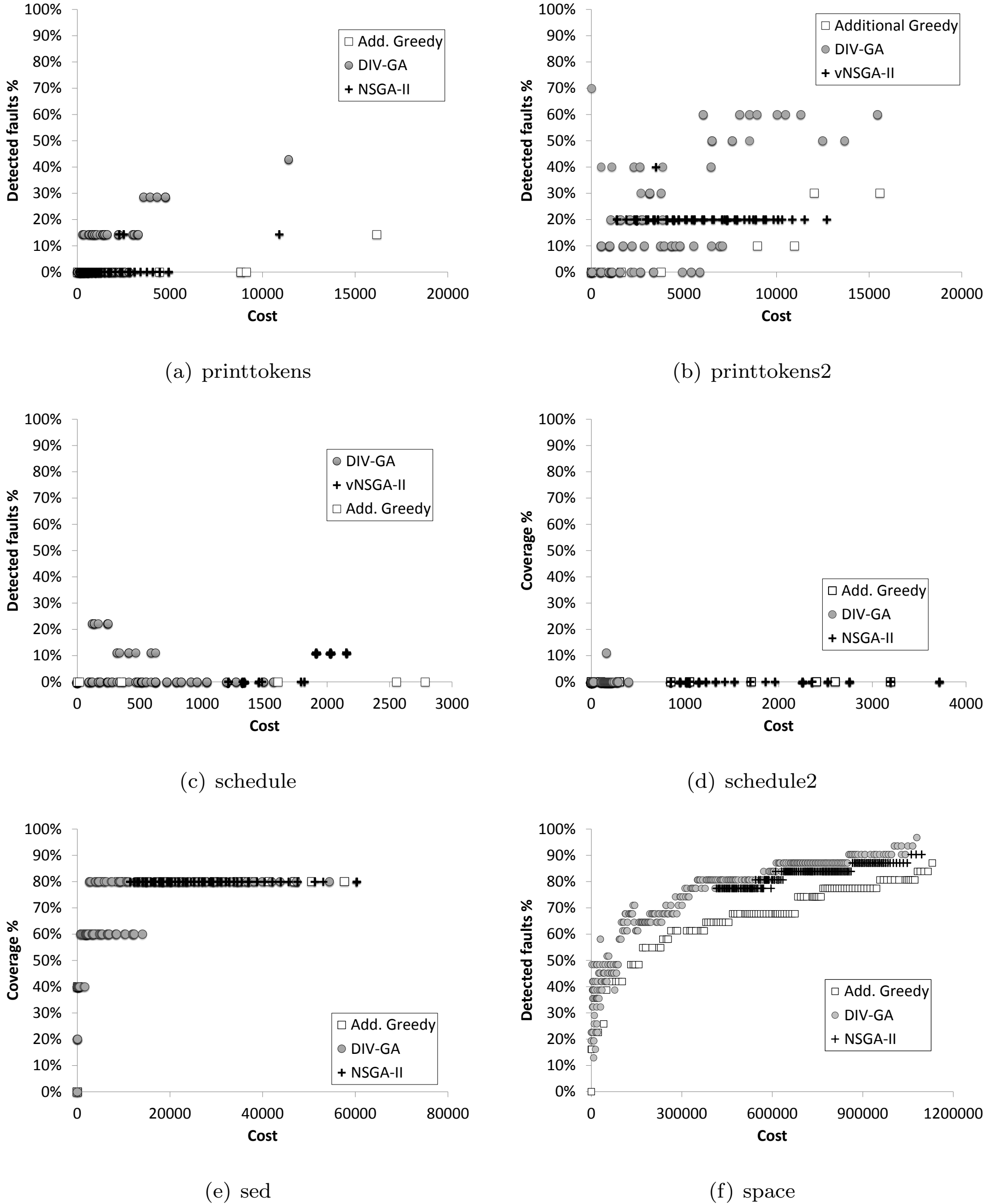
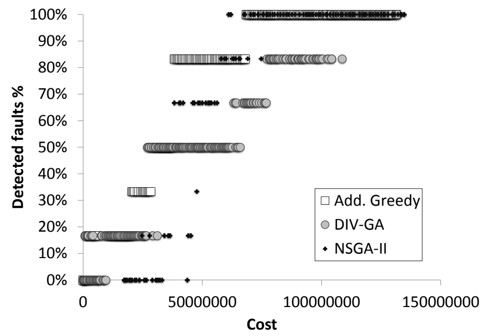


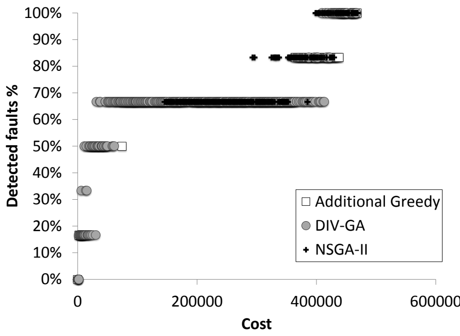
Figure C.16: Effectiveness of the achieved sub-test suites for two objectives test case selection on *printtokens*,*printtokens2*, *schedule*, *schedule*, *sed* and *space*.



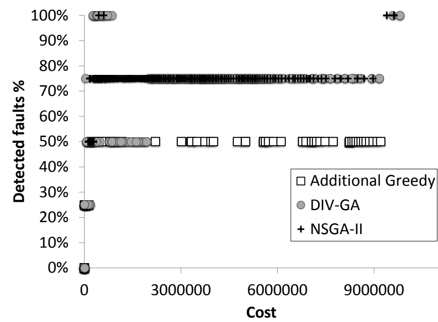
(a) vim

Figure C.17: Effectiveness of the achieved sub-test suites for two objectives test case selection on *vim*.

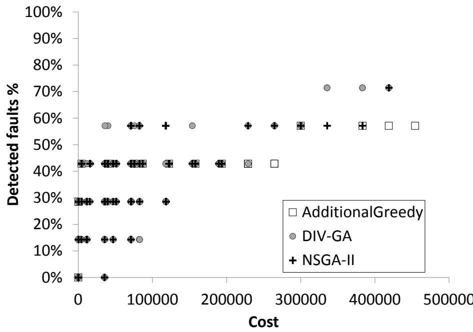
Figures C.18-C.20 plot the cost/faults curves obtained by the three algorithms. We can notice that the sub-test suites obtained by DIV-GA are able to detect more solutions than both the additional greedy and NSGA-II with a lower (or in some cases the same) execution cost. For example, on `space`, the test suites optimized by DIV-GA can detect 100% of the faults, while the percentage of faults detected by the other two algorithms is lower for the same level of execution cost. On `printtokens`, all the algorithms provide solution that are able to reveal all faults. However, DIV-GA turned out to be better than the other techniques in terms of execution cost.



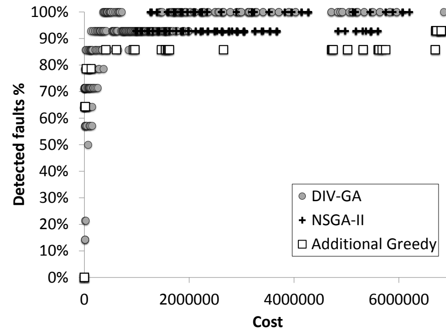
(a) bash



(b) grep

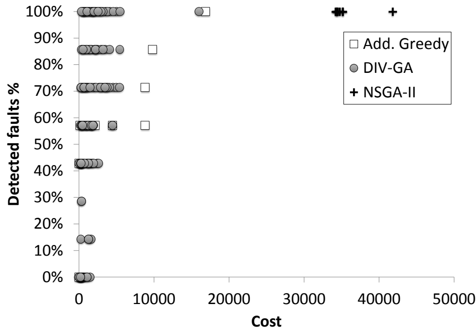


(c) gzip

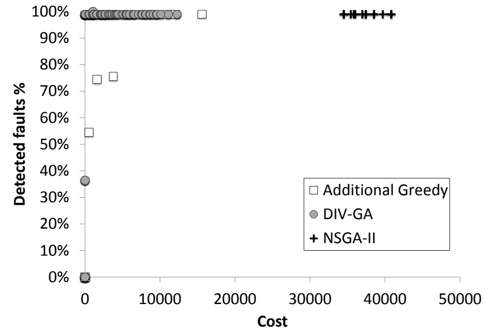


(d) flex

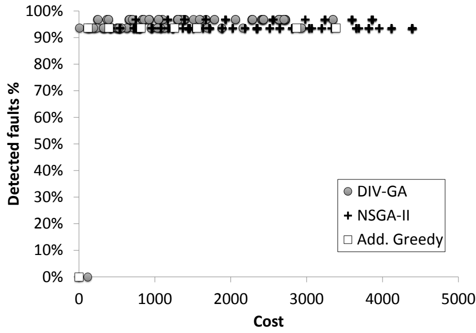
Figure C.18: Effectiveness of the achieved sub-test suites for three objectives test case selection *bash*, *flex*, *grep* and *gzip*.



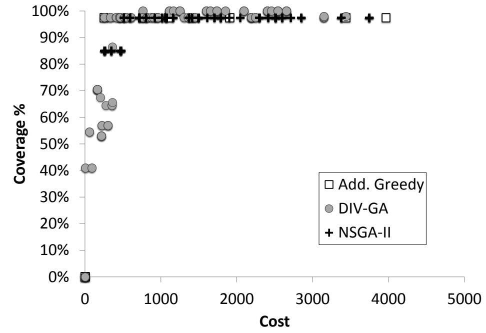
(a) printtokens



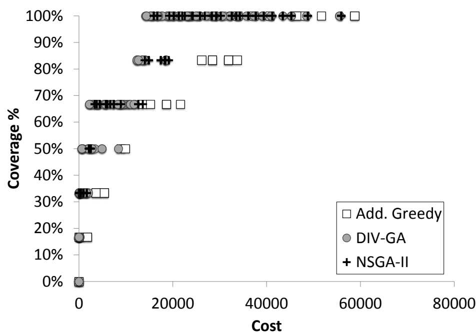
(b) printtokens2



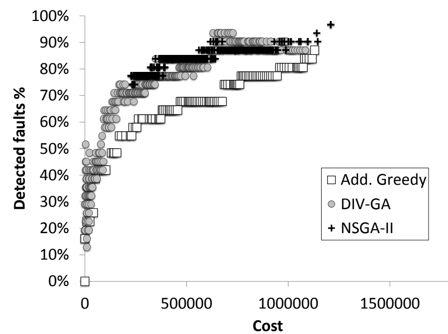
(c) schedule



(d) schedule2

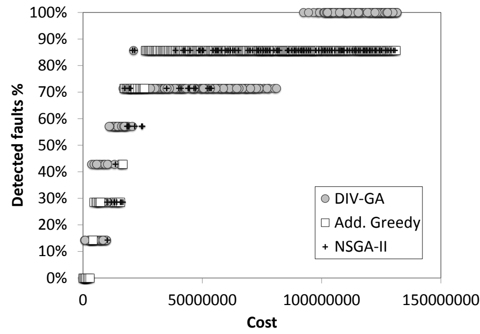


(e) sed



(f) space

Figure C.19: Effectiveness of the achieved sub-test suites for three objectives test case selection on *printtokens*,*printtokens2*, *schedule*, *schedule*, *sed* and *space*.



(a) vim

Figure C.20: Effectiveness of the achieved sub-test suites for three objectives test case selection on *vim*.

References

[1] A. Lucia, M. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: an empirical study," *Empirical Software Engineering*, pp. 1–38, 2013.

[2] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Applying a smoothing filter to improve IR-based traceability recovery processes: An empirical investigation," *Information and Software Technology*, 2012.

[3] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[4] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *35th IEEE/ACM International Conference on Software Engineering*, (San Francisco, CA, USA, May 18-26), p. to appear, 2013.

[5] F. M. Kifetew, A. Panichella, A. De Lucia, R. Oliveto, and P. Tonella, "Orthogonal exploration of the search space in evolutionary test case generation," in *Proceedings of the International Symposium on Software testing and analysis*, pp. 1–11, ACM Press, 2013.

[6] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proceedings of the International Conference on Software Testing, Verification and Validation, ICST 2013, Luxemburg, March 18-22, 2013*, IEEE, 2013.

[7] A. D. Lucia, M. D. Penta, R. Oliveto, and A. Panichella, "Estimating the evolution direction of populations to improve genetic algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference*, (Philadelphia, USA), pp. 617–624, 2012.

[8] G. Bavota, L. Colangelo, A. De Lucia, S. Fusco, R. Oliveto, and A. Panichella, "Traceme: Traceability management in eclipse," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 642–645, Sept 2012.

[9] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based traceability recovery using smoothing filters," in *Proceedings of the 19th IEEE International Conference on Program Comprehension*, (Kingston, Ontario, Canada), pp. 21–30, IEEE CS Press, 2011.

[10] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using ir methods for labeling source code artifacts: Is it worthwhile?," in *IEEE 20th International Conference on Program Comprehension (ICPC'12)*, (Passau, Germany, June 11-13), pp. 193–202, 2012.

[11] A. De Lucia, M. Di Penta, R. Oliveto, and A. Panichella, "On the role of diversity measures for multi-objective test case selection," in *Automation of Software Test (AST), 2012 7th International Workshop on*, pp. 145–151, 2012.

[12] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "When and how using structural information to improve ir-based traceability recovery," in *17th European Conference on Software Maintenance and Reengineering (CSMR), 2013*, pp. 199–208, 2013.

[13] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait la force," in *IEEE CSMR-WCRE Software Evolution Week*, 2014.

[14] B. Dit, A. Panichella, E. Moritz, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Configuring topic models for software engineering tasks in tracelab," *7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, vol. 0, pp. 105–109, 2013.

[15] G. Bavota, A. D. Lucia, R. Oliveto, A. Panichella, F. Ricci, and G. Tortora, "The role of artefact corpus in lsi-based traceability recovery," *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, vol. 0, pp. 83–89, 2013.

[16] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Traceability recovery using numerical analysis," in *Proc. of 16th Working Conference on Reverse Engineering*, (Lille, France), IEEE CS Press, 2009.

[17] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in IR-based traceability recovery," in *Proc. of 17th IEEE International Conference on Program Comprehension*, (Vancouver, British Columbia, Canada), 2009.

[18] A. Panichella, M. Oliveto, R. Di Penta, and A. De Lucia, "Improving multi-objective search based test suite optimization through diversity injection," tech. rep., University of Salerno, 2013.

[19] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Defect prediction as a multi-objective optimization problem," tech. rep., University of Salerno and University of Sannio, 2013.

[20] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[21] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, pp. 841–854, 2001.

[22] K. Tang, X. Yao, P. N. Suganthan, C. MacNish, Y. P. Chen, C. M. Chen, and Z. Yang, "Benchmark functions for the CEC 2008 special session and competition on large scale global optimization," tech. rep., Nature Inspired Computation and Applications Laboratory, USTC, China, 2007.

[23] M. Harman and B. F. Jones, "Search-based software engineering," *Information & Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[24] M. Harman and A. Mansouri, "Search based software engineering: Introduction to the special issue of the ieee transactions on software engineering," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 737–741, 2010.

[25] M. Harman, P. McMinn, J. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engineering and Verification* (B. Meyer and M. Nordio, eds.), vol. 7007 of *Lecture Notes in Computer Science*, pp. 1–59, Springer Berlin Heidelberg, 2012.

[26] B. Jones, H.-H. Sthamer, and D. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, 1996.

[27] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verif. Reliab.*, vol. 22, pp. 67–120, Mar. 2012.

[28] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, (Boston, USA), pp. 119–128, ACM Press, 2004.

[29] R. Cummins and C. O'Riordan, "Term-weighting in information retrieval using genetic programming: A three stage process," in *Proceedings of the 17th European Conference on Artificial Intelligence*, (Amsterdam, The Netherlands, The Netherlands), pp. 793–794, IOS Press, 2006.

[30] W. B. Langdon and R. Poli, *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[31] M. Lefley and M. J. Shepperd, "Using genetic programming to improve software effort estimation based on general data sets," in *Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation: PartII*, GECCO'03, pp. 2477–2487, Springer-Verlag, 2003.

[32] S. Bouktif, H. Sahraoui, and G. Antoniol, "Simulated annealing for improving software quality prediction," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pp. 1893–1900, ACM, 2006.

[33] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple hill climbing approach to software module clustering," in *Proceedings of the International Conference on Software Maintenance*, ICSM '03, IEEE Computer Society, 2003.

[34] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions Software Engineering and Methodologies*, vol. 2, pp. 270–285, 1993.

[35] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, pp. 111–123, 1995.

[36] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Inf. Process. Lett.*, vol. 60, pp. 135–141, Nov. 1996.

[37] X. Zheng and Z. Liu, "Based on particle swarm engineering project progress control," in *Software Engineering and Knowledge Engineering: Theory and Practice* (Y. Wu, ed.), vol. 114 of *Advances in Intelligent and Soft Computing*, pp. 69–75, Springer Berlin Heidelberg, 2012.

[38] W.-N. Chen and J. Zhang, "Ant colony optimization for software project scheduling and staffing with an event-based scheduler," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1–17, Jan 2013.

[39] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *Software Engineering, IEEE Transactions on*, vol. SE-2, pp. 223–226, Sept 1976.

[40] I. S. Staff, "Changing face of software engineering," *IEEE Software*, vol. 11, no. 1, pp. 4–5, 1994.

[41] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Empirical software engineering and verification," ch. Search Based Software Engineering: Techniques, Taxonomy, Tutorial, pp. 1–59, Berlin, Heidelberg: Springer-Verlag, 2012.

[42] M. Harman, "The relationship between search based software engineering and predictive modeling," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE 2010, Timisoara, Romania, September 12-13, 2010*, p. 1, ACM, 2010.

[43] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*, pp. 342–357, IEEE Computer Society, 2007.

[44] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, 2012.

[45] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, pp. 105–156, 2004.

[46] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, 2011.

[47] A. Baresel, D. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 108–118, ACM, 2004.

[48] A. Baresel, H. Sthamer, and M. Schmidt, "Fitness function design to improve evolutionary structural testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pp. 1329–1336, Morgan Kaufmann Publishers Inc., 2002.

[49] L. Bottaci, "Instrumenting programs with flag variables for test data search by genetic algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pp. 1337–1342, Morgan Kaufmann Publishers Inc., 2002.

[50] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 3–16, 2004.

[51] A. Bagnall, V. Rayward-Smith, and I. Whittley, "The next release problem," *Information and Software Technology*, vol. 43, no. 14, pp. 883 – 890, 2001.

[52] J. S. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and M. Toro, "An evolutionary approach to estimating software development projects," *Information and Software Technology*, vol. 43, no. 14, pp. 875 – 882, 2001.

[53] G. Antoniol, M. D. Penta, and M. Harman, "A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty," in *Proceedings of the Software Metrics, 10th International Symposium*, METRICS '04, pp. 172–183, IEEE Computer Society, 2004.

[54] G. Antoniol, M. Di Penta, and M. Harman, "Search-based techniques applied to optimization of project planning for a massive maintenance project," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pp. 240–249, IEEE Computer Society, 2005.

[55] M. Harman, R. M. Hierons, and M. Proctor, "A new representation and crossover operator for search-based optimization of software modularization," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pp. 1351–1358, Morgan Kaufmann Publishers Inc., 2002.

[56] B. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 193–208, March 2006.

[57] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An approach for qos-aware service composition based on genetic algorithms," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, GECCO '05, (New York, NY, USA), pp. 1069–1075, ACM, 2005.

[58] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 1–9, ACM, 1999.

[59] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 1066–1071, ACM, 2011.

[60] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux, "Flopsy: Search-based floating point constraint solving for symbolic execution," in *Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems*, ICTSS'10, pp. 142–157, 2010.

[61] S. Yoo, R. Nilsson, and M. Harman, "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, 2011.

[62] S. Comford, M. Feather, J. Dunphy, J. Salcedo, and T. Menzies, "Optimizing spacecraft design optimization engine development: progress and plans," in *Proceedings of the IEEE Aerospace Conference*, vol. 8, pp. 3681–3690, 2003.

[63] P. Baker, M. Harman, K. Steinhofel, and A. Skaliotis, "Search based approaches to component selection and prioritization for the next release problem," in *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pp. 176–185, 2006.

[64] Y. Zhang, E. Alba, J. J. Durillo, S. Eldh, and M. Harman, "Today/future importance analysis," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, (New York, NY, USA), pp. 1357–1364, ACM, 2010.

[65] S. Yoo, M. Harman, and S. Ur, "Highly scalable multi objective test suite minimisation using graphics cards," in *Proceedings of the Third international conference on Search based software engineering*, pp. 219–236, Springer-Verlag, 2011.

[66] F. Asadi, G. Antoniol, and Y. Guèhèneuc, "Concept location with genetic algorithms: A comparison of four distributed architectures," in *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, pp. 153–162, Sept 2010.

[67] P. McMinn, “Search-based software testing: Past, present and future,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pp. 153–163, 2011.

[68] S. Yoo and M. Harman, “Pareto efficient multi-objective test case selection,” in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, (London, UK), pp. 140–150, ACM Press, 2007.

[69] G. Fraser and A. Arcuri, “Sound empirical evidence in software testing,” in *ICSE*, pp. 178–188, 2012.

[70] M.-A. D. Storey, “Theories, tools and research methods in program comprehension: past, present and future,” *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[71] T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: a study of developer work habits,” in *Proceedings of the 28th International Conference on Software Engineering*, (Shanghai, China), pp. 492–501, ACM Press, 2006.

[72] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks,” *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[73] D. Poshyvanyk, Y. Gael-Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval,” *IEEE Trans. on Softw. Eng.*, vol. 33, no. 6, pp. 420–432, 2007.

[74] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, “Recovering traceability links between code and documentation,” *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[75] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, “Integrated impact analysis for managing software changes,” in *Proc. of the 34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, (Zurich, Switzerland, June 2-9), pp. 430–440, 2012.

[76] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *Proc. of 29th IEEE/ACM International Conference on Software Engineering (ICSE'07)*, (Minneapolis, Minnesota, USA), pp. 499–510, 2007.

[77] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, “Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?,” in *Proceedings of the 28th International Conference on Software Maintenance*, (Riva del Garda, Italy), IEEE CS Press, 2012.

[78] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by latent semantic analysis,” *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[79] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[80] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *Proc. of the 18th IEEE International Conference on Program Comprehension*, (Braga, Portugal), pp. 68–71, 2010.

[81] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proc.*, pp. 33–47, Springer, 2011.

[82] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *Proc. of the 34th IEEE/ACM International Conference on Software Engineering (ICSE'12), Zurich, Switzerland, June 2-9*, pp. 837–847, 2012.

[83] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *28th International Conference on Software Engineering (ICSE 2006)*, (Shanghai, China, May 20-28, 2006), pp. 452–461, ACM, 2006.

[84] B. Turhan, T. Menzies, A. B. Bener, and J. S. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[85] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 91–100, ACM, 2009.

[86] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. R. Cok, "Local vs. global models for effort estimation and defect prediction," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011*, pp. 343–351, IEEE, 2011.

[87] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *9th IEEE Working Conference o Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pp. 60–69, IEEE, 2012.

[88] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *Proceedings of the ACM-Sigsoft 20th International Symposium on the Foundations of Software Engineering (FSE-20)*, (Research Triangle Park, NC, USA), p. 61, ACM, 2012.

[89] E. Arisholm and L. C. Briand, “Predicting fault-prone components in a java legacy system,” in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE ’06, pp. 8–17, ACM, 2006.

[90] E. Arisholm, L. C. Briand, and E. B. Johannessen, “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models,” *J. Syst. Softw.*, vol. 83, pp. 2–17, January 2010.

[91] A. Rogers and A. Prügel-Bennett, “Genetic drift in genetic algorithm selection schemes,” *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 298–303, 1999.

[92] M. Alshraideh, L. Bottaci, and B. A. Mahafzah, “Using program data-state scarcity to guide automatic test data generation,” *Software Quality Journal*, vol. 18, no. 1, pp. 109–144, 2010.

[93] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, “Searching for cognitively diverse tests: Towards universal test diversity metrics,” in *Software Testing Verification and Validation Workshop, 2008. ICSTW’08. IEEE International Conference on*, pp. 178–186, IEEE, 2008.

[94] G. Rothermel and M. J. Harrold, “Empirical studies of a safe regression test selection technique,” *IEEE Transactions on Software Engineering*, vol. 24, pp. 401–419, 1998.

[95] V. Garousi and T. Varma, “A replicated survey of software testing practices in the canadian province of alberta: What has changed from 2004 to 2009?,” *J. Syst. Softw.*, vol. 83, pp. 2251–2262, Nov. 2010.

[96] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Prioritizing test cases for regression testing,” *Software Engineering Notes*, vol. 25, pp. 102–112, 2000.

[97] J. Black, E. Melachrinoudis, and D. Kaeli, “Bi-criteria models for all-uses test suite reduction,” in *Proceedings of the 26th International Conference on Software Engineering*, ICSE ’04, (Washington, DC, USA), pp. 106–115, IEEE Computer Society, 2004.

[98] G. Rothermel and M. Harrold, “A safe, efficient algorithm for regression test selection,” in *Software Maintenance ,1993. CSM-93, Proceedings., Conference on*, pp. 358–367, 1993.

[99] K. Fischer, “A test case selection method for the validation of software maintenance modifications,” in *Proceedings of COMPSAC*, (New York), pp. 421–426, IEEE, 1977.

[100] A. Srivastava and J. Thiagarajan, “Effectively prioritizing tests in development environment,” in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 97–106, ACM, 2002.

[101] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*, (Washington, DC, USA), pp. 329–338, IEEE Computer Society, 2001.

[102] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," tech. rep., Department of Computer Science and Engineering, 2006.

[103] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, vol. 83, no. 4, pp. 689–701, 2010.

[104] D. Jeffrey, "Test suite reduction with selective redundancy," in *IEEE International Conference on Software Maintenance (ICSM) 2005*, pp. 549–558, IEEE Computer Society, 2005.

[105] S. Sampath, R. Bryce, and A. Memon, "A uniform representation of hybrid criteria for regression testing," *Software Engineering, IEEE Transactions on*, vol. 39, no. 10, pp. 1326–1344, 2013.

[106] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, vol. 26, pp. 369–395, 2004.

[107] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast elitist multi-objective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.

[108] D. C. Montgomery, *Design amd Analysis of Experiments.* Wiley, 3rd ed., 2009.

[109] A. De Lucia, M. Di Penta, R. Oliveto, and A. Panichella, "Estimating the evolution direction of populations to improve genetic algorithms," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pp. 617–624, ACM, 2012.

[110] A. Auger, J. Bader, D. Brockhoff, and E. Zitzler, "Theory of the hypervolume indicator: optimal μ-distributions and the choice of the reference point," in *Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms*, FOGA '09, pp. 87–102, ACM, 2009.

[111] T. Mens and S. Demeyer, eds., *Software Evolution.* Springer, 2008.

[112] P. Grubb and A. A. Takang, *Software Maintenance: Concepts and Practice.* World Scientific Publishing Co. Pte. Ltd., 2st ed., Jan. 2003.

[113] T. A. Standish, "An essay on software reuse," *IEEE Transaction on Software Engineering*, vol. SE-10, no. 5, pp. 494–497, 1984.

[114] R. Glass, "Frequently forgotten fundamental facts about software engineering," *Software, IEEE*, vol. 18, pp. 112–111, May 2001.

[115] M. Dorfman and R. H. Thayer, *Software engineering.* IEEE, 1996.

[116] P. Chittimalli and M. Harrold, "Recomputing coverage information to assist regression testing," *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 452–469, 2009.

[117] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pp. 50–, IEEE Computer Society, 1999.

[118] M. Harman, S. Swift, and K. Mahdavi, "An empirical study of the robustness of two module clustering fitness functions," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, GECCO '05, pp. 1029–1036, ACM, 2005.

[119] T. Bodhuin, M. Di Penta, and L. Troiano, "A search-based approach for dynamically re-packaging downloadable applications," in *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pp. 27–41, IBM Corp., 2007.

[120] S. Huynh and Y. Cai, "An evolutionary approach to software modularity analysis," in *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, ACoM '07, pp. 6–, IEEE Computer Society, 2007.

[121] M. Cohen, S. B. Kooi, and W. Srisa-an, "Clustering the heap in multi-threaded applications for improved garbage collection," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pp. 1901–1908, ACM, 2006.

[122] C. Del Rosso, "Reducing internal fragmentation in segregated free lists using genetic algorithms," in *Proceedings of the 2006 International Workshop on Workshop on Interdisciplinary Software Engineering Research*, WISER '06, pp. 57–60, ACM, 2006.

[123] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[124] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pp. 1106–1113, ACM, 2007.

[125] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pp. 131–142, ACM, 2007.

[126] M. Stephenson, U.-M. O'Reilly, M. C. Martin, and S. Amarasinghe, "Genetic programming applied to compiler heuristic optimization," in *Proceedings of the 6th European Conference on Genetic Programming*, EuroGP'03, pp. 238–253, Springer-Verlag, 2003.

[127] K. Hoste and L. Eeckhout, "Cole: Compiler optimization level exploration," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pp. 165–174, ACM, 2008.

[128] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving transformation sequences using genetic algorithms," in *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, SCAM '04, pp. 66–75, IEEE Computer Society, 2004.

[129] M. Kessentini, H. Sahraoui, and M. Boukadoum, "Model transformation as an optimization problem," in *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pp. 159–173, Springer-Verlag, 2008.

[130] A. Nisbet, "Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation," in *High-Performance Computing and Networking* (P. Sloot, M. Bubak, and B. Hertzberger, eds.), vol. 1401 of *Lecture Notes in Computer Science*, pp. 987–989, Springer Berlin Heidelberg, 1998.

[131] I. Bate and P. Emberson, "Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pp. 221–230, IEEE Computer Society, 2006.

[132] H. Sahraoui, P. Valtchev, I. Konkobo, and S. Shen, "Object identification in legacy code as a grouping problem," in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pp. 689–696, 2002.

[133] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pp. 947–954, ACM, 2009.

[134] A. Hani, A. S. Houari, S. Osama, A. Nicolas, and S. Ducasse, "Towards automatically improving package structure while respecting original design decisions," in *WCRE*, pp. 212–221, 2013.

[135] C. A. Coello Coello, G. B. Lamont, and D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[136] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[137] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 1st ed., 2007.

[138] T. Bäck, D. B. Fogel, and Z. Michalewicz, *Evolutionary Computation 2: Advanced Algorithms and Operators*. IOP Publishing Ltd, first ed., 2000.

[139] J. Antonisse, "A new interpretation of schema notation that overturns the binary encoding constraint," in *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 86–91, Morgan Kaufmann Publishers Inc., 1989.

[140] L. Davis, *Handbook of Genetic Algorithms*. International Thomson Computer Press, 1996.

[141] T. Nomura, "An analysis on crossovers for real number chromosomes in an infinite population size," in *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2*, IJCAI'97, (San Francisco, CA, USA), pp. 936–941, Morgan Kaufmann Publishers Inc., 1997.

[142] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, 2012.

[143] J. D. Schaffer, "Multiple objective optimization with vector evaluated genetic algorithms," in *Proceedings of the 1st International Conference on Genetic Algorithms*, (Hillsdale, NJ, USA), pp. 93–100, L. Erlbaum Associates Inc., 1985.

[144] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[145] N. Srinivas and K. Deb, "Muiltiobjective optimization using nondominated sorting in genetic algorithms," *Evol. Comput.*, vol. 2, pp. 221–248, Sept. 1994.

[146] J. Horn, N. Nafpliotis, and D. Goldberg, "A niched pareto genetic algorithm for multiobjective optimization," in *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pp. 82–87 vol.1, 1994.

[147] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the 10th International Workshop on Program Comprehension*, (Paris, France), pp. 271–280, IEEE Computer Society, 2002.

[148] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling, "Fast collapsed gibbs sampling for latent dirichlet allocation," in *Proc. of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 569–577, ACM, 2008.

[149] D. Binkley and D. Lawrie, "Information retrieval applications in software maintenance and evolution," *Encyclopedia of Software Engineering*, 2009.

[150] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information retrieval models for recovering traceability links between code and documentation," in *Proc. of 16th IEEE International Conference on Software Maintenance*, (San Jose, California, USA), pp. 40–51, IEEE CS Press, 2000.

[151] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods.," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[152] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artefact management systems using information retrieval methods," *ACM Trans. on Soft. Eng. and Methodology*, vol. 16, no. 4, 2007.

[153] A. Marcus, J. I. Maletic, and A. Sergeyev, "Recovery of traceability links between software documentation and source code," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 5, pp. 811–836, 2005.

[154] M. Lormans and A. van Deursen, "Can lsi help reconstructing requirements traceability in design and test?," in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pp. 10–56, 2006.

[155] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proc. of 11th Working Conference on Reverse Engineering*, (Delft, The Netherlands), pp. 214–223, IEEE CS Press, 2004.

[156] D. Lawrie, H. Feild, and D. Binkley, "Leveraged quality assessment using information retrieval techniques," in *Proc. of 14th IEEE International Conference on Program Comprehension*, (Athens, Greece), pp. 149–158, IEEE CS Press, 2006.

[157] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transaction on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[158] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proc. of 22nd IEEE International Conference on Software Maintenance*, (Philadelphia, PA, USA), pp. 469 – 478, IEEE CS Press, 2006.

[159] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, “Using information retrieval based coupling measures for impact analysis,” *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[160] D. Poshyvanyk and A. Marcus, “Using traceability links to assess and maintain the quality of software documentation,” in *Proc. of International Symposium on Grand Challenges in Traceability*, (Lexington, Kentuky, USA), pp. 27–30, ACM Press, 2007.

[161] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in *30th IEEE/ACM International Conference on Software Engineering*, (Leipzig, Germany, May 10-18), pp. 461–470, 2008.

[162] E. Enslen, E. Hill, L. L. Pollock, and K. Vijay-Shanker, “Mining source code to automatically split identifiers for software analysis,” in *Proc. of the 6th International Working Conference on Mining Software Repositories*, (Vancouver, British Columbia, Canada), pp. 71–80, 2009.

[163] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, “Tidier: an identifier splitting approach using speech recognition techniques,” *Journal of Software Maintenance and Evolution: Research and Practice*, p. in press, 2011.

[164] D. J. Lawrie, D. Binkley, and C. Morrell, “Normalizing source code vocabulary,” in *Proc. of the 17th Working Conference on Reverse Engineering*, (Beverly, MA, USA), pp. 3–12, IEEE CS Press, 2010.

[165] D. Lawrie and D. Binkley, “Expanding identifiers to normalize source code vocabulary,” in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, (Williamsburg, VA, USA), pp. 113 –122, IEEE CS Press, 2011.

[166] N. Madani, L. Guerrouj, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “Recognizing words from source code identifiers using speech recognition techniques,” in *Proc. of the 14th European Conference on Software Maintenance and Reengineering*, (Madrid, Spain), 2010.

[167] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval.* Addison-Wesley, 1999.

[168] D. K. Harman, “Overview of the first Text REtrieval Conference (TREC-1),” in *Proc. of the First Text REtrieval Conference (TREC-1)*, pp. 1–20, NIST Special Publication, 1993.

[169] M. F. Porter, “An algorithm for suffix stripping,” *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[170] S. T. Dumais, “Improving the retrieval of information from external sources,” *Behavior Research Methods, Instruments and Computers*, vol. 23, pp. 229–236, 1991.

[171] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing supporting evidence to improve dynamic requirements traceability," in *Proc. of 13th IEEE International Requirements Engineering Conference*, (Paris, France), pp. 135–144, IEEE CS Press, 2005.

[172] G. Salton, A. Wong, and C. S. Yang, "A vector space model for information retrieval," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[173] H. U. Asuncion, A. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering*, (Cape Town, South Africa), pp. 95–104, ACM Press, 2010.

[174] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in *Proc. of 16th IEEE International Conference on Program Comprehension*, (Amsterdam, the Netherlands), pp. 103–112, IEEE CS Press, 2008.

[175] J. H. Hayes, A. Dekhtyar, and J. Osborne, "Improving requirements tracing via information retrieval," in *Proc. of 11th IEEE International Requirements Engineering Conference*, (Monterey, California, USA), pp. 138–147, IEEE CS Press, 2003.

[176] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," in *Proc. of 22nd IEEE International Conference on Software Maintenance*, (Philadelphia, PA), pp. 299–309, IEEE CS Press, 2006.

[177] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. of 25th International Conference on Software Engineering*, (Portland, Oregon, USA), pp. 125–135, IEEE CS Press, 2003.

[178] R. Settimi, J. Cleland-Huang, O. Ben Khadra, J. Mody, W. Lukasik, and C. De Palma, "Supporting software evolution through dynamically retrieving traces to UML artifacts," in *Proc. of 7th IEEE International Workshop on Principles of Software Evolution*, (Kyoto, Japan), pp. 49–54, IEEE CS Press, 2004.

[179] S. Yadla, J. Huffman Hayes, and A. Dekhtyar, "Tracing requirements to defect reports: An application of information retrieval techniques," *Innovations in Systems and Software Engineering: A NASA Journal*, vol. 1, no. 2, pp. 116–124, 2005.

[180] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, vol. 1, ch. Real rectangular matrices. Boston: Birkhauser, 1998.

[181] M. Lormans, A. Deursen, and H.-G. Gross, "An industrial case study in reconstructing requirements views," *Empirical Software Engineering*, vol. 13, no. 6, pp. 727–760, 2008.

[182] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proc. of 23rd International Conference on Software Engineering*, (Toronto, Ontario, Canada), pp. 103–112, IEEE CS Press, 2001.

[183] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information & Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[184] N. Kaushik and L. Tahvildari, "A comparative study of the performance of ir models on duplicate bug detection," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 159–168, 2012.

[185] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Tracing object-oriented code into functional requirements," in *Proc. of 8th IEEE International Workshop on Program Comprehension*, (Limerick, Ireland), pp. 79–87, IEEE CS Press, 2000.

[186] G. Antoniol, G. Casazza, and A. Cimitile, "Traceability recovery by modelling programmer behaviour," in *Proc. of 7th Working Conference on Reverse Engineering*, vol. 240-247, (Brisbane, Queensland, Australia), IEEE CS Press, 2000.

[187] M. Di Penta, S. Gradara, and G. Antoniol, "Traceability recovery in RAD software systems," in *Proc. of 10th International Workshop in Program Comprehension*, (Paris, France), pp. 207–216, IEEE CS Press, 2002.

[188] D. Jurafsky and J. Martin, *Speech and Language Processing.* Prentice Hall, 2000.

[189] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *Proc. of ICSE*, pp. 155–164, 2010.

[190] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards mining replacement queries for hard-to-retrieve traces," in *Proc. of ASE*, pp. 245–254, 2010.

[191] X. Zou, R. Settimi, and J. Cleland-Huang, "Improving automated requirements trace retrieval: a study of term-based enhancement methods," *Empirical Software Engineering*, vol. 15, no. 2, pp. 119–146, 2010.

[192] X. Zou, R. Settimi, and J. Cleland-Huang, "Term-based enhancement factors for improving automated requirement trace retrieval," in *Proc. of International Symposium on Grand Challenges in Traceability*, (Lexington, Kentuky, USA), pp. 40–45, ACM Press, 2007.

[193] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[194] M. Eaddy, A. Aho, G. Antoniol, and Y.-G. Gueheneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 53–62, June 2008.

[195] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proc. of 22nd IEEE/ACM International Conference on Automated Software Engineering*, (Atlanta, Georgia, USA), ACM Press, 2007.

[196] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a static non-interactive approach to feature location," *ACM Transactions on Software Engineering and Methodologies*, vol. 15, no. 2, pp. 195–226, 2006.

[197] D. Poshyvanyk and D. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *Proc. of 15th IEEE International Conference on Program Comprehension*, (Banff, Alberta, Canada), pp. 37–48, IEEE CS Press, 2007.

[198] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 351–360, Sept 2009.

[199] D. Lawrie, C. Uehlinger, and D. Binkley, "Vocabulary normalization improves ir-based concept location," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, IEEE Computer Society, 2012.

[200] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?," in *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pp. 11–20, IEEE Computer Society, 2011.

[201] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, pp. 52–61, IEEE Computer Society, 2008.

[202] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pp. 366–374, Nov 2010.

[203] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 45–54, ACM, 2010.

[204] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. of the 17th Working Conference on Reverse Engineering*, (Beverly, MA, USA), pp. 35–44, IEEE Computer Society, 2010.

[205] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010*, pp. 55–64, IEEE Computer Society, 2010.

[206] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Codetopics: which topic am i coding now?," in *Proc. of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pp. 1034–1036, ACM, 2011.

[207] P. Baldi, C. V. Lopes, E. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," in *Proc. of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*, (Nashville, TN, USA), pp. 543–562, 2008.

[208] E. Linstead, C. V. Lopes, and P. Baldi, "An application of Latent Dirichlet Allocation to analyzing software evolution," in *Proc. of the 7th International Conference on Machine Learning and Applications*, (San Diego, California, USA), pp. 813–818, IEEE CS Press, 2008.

[209] S. Medini, G. Antoniol, Y.-G. Guéhéneuc, M. Di Penta, and P. Tonella, "Scan: an approach to label and relate execution trace segments," in *Proceedings of the 19th Working Conference on Reverse Engineering*, (Kingston, Ontario, Canada), IEEE Press, 2012.

[210] D. Falessi, G. Cantone, and G. Canfora, "Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 1, pp. 18–44, 2013.

[211] S. Grant and J. R. Cordy, "Estimating the optimal number of latent concepts in source code analysis," in *Proc. of the 10th International Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, pp. 65–74, 2010.

[212] R. Cummins, *The Evolution and Analysis of Term-Weighting Schemes in Information Retrieval.* PhD thesis, National University of Ireland, 2008.

[213] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proc. of the National Academy of Sciences*, vol. 101, no. Suppl. 1, pp. 5228–5235, 2004.

[214] Y. Teh, M. Jordan, M. Beal, and D. Blei, "Hierarchical Dirichlet processes," *Journal of the American Statistical Association*, vol. 101, no. 476, pp. 1566–1581, 2006.

[215] L. Biggers, C. Bocovich, R. Capshaw, B. Eddy, L. Etzkorn, and N. Kraft, "Configuring Latent Dirichlet Allocation based feature location," *Empirical Software Engineering (EMSE)*, p. to appear, 2012.

[216] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using Latent Dirichlet Allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[217] A. Nguyen, T. Nguyen, J. Al-Kofahi, H. Nguyen, and T. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, (Lawrence, Kansas, USA, November 6-10), pp. 263–272, IEEE, 2011.

[218] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining eclipse developer contributions via author-topic models," in *Proc. of 4th International Workshop on Mining Software Repositories*, (Minneapolis, Minnesota, USA), pp. 30–33, IEEE CS Press, 2007.

[219] M. Gethers, R. Oliveto, D. Poshyvanyk, and A.De Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *Proc. of the 27th International Conference on Software Maintenance (ICSM'11)*, (Williamsburg, VA, USA, Sept. 25-Oct. 1), pp. 133–142, IEEE Press, 2011.

[220] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models." Forthcoming in *Empirical Software Engineering*, 2013.

[221] A. J. Hindle, M. W. Godfrey, and R. C. Holt, "What's hot and what's not: Windowing developer topic analysis," in *Proc. of the 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, Canada, September 20-26*, 2009.

[222] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[223] J. Kogan, *Introduction to Clustering Large and High-Dimensional Data.* New York, NY, USA: Cambridge University Press, 2007.

[224] R Core Team, *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[225] V. Basili, G. Caldiera, and D. H. Rombach, *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering.* John Wiley and Sons, 1994.

[226] W. J. Conover, *Practical Nonparametric Statistics.* Wiley, 3rd edition ed., 1998.

[227] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70, 1979.

[228] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering (EMSE)*, pp. 1–33, to. appear, 2012.

[229] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pp. 1–10, 2010.

[230] Y. Collette and P. Siarry, *Multiobjective Optimization: Principles and Case Studies.* Springer, 2004.

[231] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[232] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th International Conference on Software Engineering (ICSE 2008)*, (Leipzig, Germany, May), pp. 181–190, ACM, 2008.

[233] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[234] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. of 29th International Conference on Software Engineering*, (Minneapolis, Minnesota, USA), pp. 489–498, IEEE Computer Society, 2007.

[235] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Software Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[236] B. L., B. V., and H. C., "Developing interpretable models with optimized set reduction for identifying high-risk software components," *IEEE Trans. on Software Engineering*, vol. 19, no. 11, pp. 1028–1044, 1993.

[237] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker, "Does calling structure information improve the accuracy of fault prediction?," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pp. 61–70, May 2009.

[238] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y. Gueheneuc, and G. Antoniol, "Physical and conceptual identifier dispersion: Measures and relation to fault proneness," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–5, Sept 2010.

[239] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pp. 2–12, ACM, 2008.

[240] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 1–11, IEEE Computer Society, 2009.

[241] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?," in *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, FASE'10, pp. 59–73, Springer-Verlag, 2010.

[242] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[243] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction.," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[244] T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pp. 435–438, 2009.

[245] M. Bezerra, A. L. I. Oliveira, and S. R. L. Meira, "A constructive rbf neural network for estimating the probability of defects in software modules," in *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pp. 2869–2874, 2007.

[246] E. Ceylan, F. Kutlubay, and A. Bener, "Software defect identification using machine learning techniques," in *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*, pp. 240–247, 2006.

[247] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using bayesian nets," *Information and Software Technology*, vol. 49, no. 1, pp. 32 – 43, 2007.

[248] A. Okutan and O. Yildiz, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, pp. 1–28, 2012.

[249] A. Nelson, T. Menzies, and G. Gay, "Sharing experiments using open-source software," *Software: Practice and Experience*, vol. 41, no. 3, pp. 283–305, 2011.

[250] Y. Liu, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 852–864, Nov. 2010.

[251] N. Fenton, P. Krause, and M. Neil, "Software measurement: uncertainty and causal modeling," *Software, IEEE*, vol. 19, no. 4, pp. 116–122, 2002.

[252] M. D. Buhmann and M. D. Buhmann, *Radial Basis Functions.* New York, NY, USA: Cambridge University Press, 2003.

[253] L. C. Briand, W. L. Melo, and J. Würst, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering*, vol. 28, pp. 706–720, 2002.

[254] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement (ESEM 2009)*, (Lake Buena Vista, Florida, USA), pp. 460–463, 2009.

[255] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 382–391, IEEE Press, 2013.

[256] B. Turhan, A. T. Misirli, and A. B. Bener, "Empirical evaluation of mixed-project defect prediction models," in *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, (Oulu, Finland), pp. 396–403, IEEE, 2011.

[257] J. L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*. Duxbury, 1999.

[258] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pp. 119–125, ACM, 2006.

[259] L. Rokach and O. Maimon, "Decision trees," in *The Data Mining and Knowledge Discovery Handbook*, pp. 165–192, Springer-Verlag New York, Inc., 2005.

[260] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.

[261] MATLAB, *version 7.10.0 (R2010b)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[262] M. Stone, "Cross-validatory choice and assesment of statistical predictions (with discussion)," *Journal of the Royal Statistical Society B*, vol. 36, pp. 111–147, 1974.

[263] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[264] T. Yang, J. Liu, L. Mcmillan, and W. Wang, "A fast approximation to multidimensional scaling, by," in *Proceedings of the ECCV Workshop on Computation Intensive Methods for Computer Vision (CIMCV*, 2006.

[265] A. Arcuri and L. C. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pp. 1–10, ACM, 2011.

[266] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *Software Engineering, IEEE Transactions on*, vol. 36, pp. 593–617, Sept 2010.

[267] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pp. 384–396, ACM, 1993.

[268] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification, and Reliability*, vol. 12, pp. 219–249, 2002.

[269] S. Mcmaster and A. M. Memon, "Call stack coverage for test suite reduction," in *IEEE International Conference on Software Maintenance (ICSM) 2005*, pp. 539–548, IEEE Computer Society, 2005.

[270] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Softw. Eng.*, vol. 22, pp. 529–551, Aug. 1996.

[271] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time aware test suite prioritization," in *Proceedings of the International Symposium on Software testing and analysis*, pp. 1–12, ACM Press, 2006.

[272] J. C. King, "A new approach to program testing," in *Proceedings of the International Conference on Reliable Software*, (New York, NY, USA), pp. 228–233, ACM, 1975.

[273] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[274] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pp. 195–200, ACM, 1996.

[275] R. A. DeMillo and A. J. Offutt, "Experimental results from an automatic test case generator," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 2, pp. 109–127, 1993.

[276] B. Korel, "Automated test data generation for programs with procedures," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pp. 209–215, ACM, 1996.

[277] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, 1996.

[278] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. 2, no. 3, pp. 215–222, 1976.

[279] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[280] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 438–444, 1984.

[281] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, pp. 60–66, Jan. 2010.

[282] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pp. 265–275, ACM, 2011.

[283] M. Roper, "Computer aided software testing using genetic algorithms," in *Proceedings of the 10th International Quality Week*, 1997.

[284] A. Watkins, "The automatic generation of test data using genetic algorithms," in *Proceedings of the Fourth Software Quality Conference*, pp. 300–309, 1995.

[285] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, A. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," in *5th International Conference on Software Engineering and its Applications*, pp. 625–636, ACM, 1992.

[286] S. Wappler and F. Lammermann, "Using evolutionary algorithms for the unit testing of object-oriented software," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, GECCO '05, pp. 1053–1060, ACM, 2005.

[287] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (Szeged, Hungary), pp. 416–419, 2011.

[288] N. Tracey, J. Clark, and Mander, "The way forward for unifying dynamic test-case generation: The optimisation-based approach," in *International Workshop on Dependable Computing and Its Applications*, pp. 169–180, Dept of Computer Science, 1998.

[289] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, ASE '98, pp. 285–, IEEE Computer Society, 1998.

[290] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research articles," *Softw. Test. Verif. Reliab.*, vol. 16, no. 3, pp. 175–203, 2006.

[291] H. Hemmati, A. Arcuri, and L. Briand, "Reducing the cost of model-based testing through test case diversity," in *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, pp. 63–78, Springer-Verlag, 2010.

[292] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1990.

[293] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[294] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 159–182, Feb. 2002.

[295] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, "A framework of greedy methods for constructing interaction test suites," in *International Conference on Software Engineering (ICSE05*, pp. 146–155, 2005.

[296] M. Cohen, M. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *Software Engineering, IEEE Transactions on*, vol. 34, pp. 633–650, 2008.

[297] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Empirical Software Engineering, 2005. International Symposium on*, 2005.

[298] M. Marré and A. Bertolino, "Using spanning sets for coverage testing," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 974–984, Nov. 2003.

[299] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proceedings of the International Conference on Software Maintenance*, pp. 34–44, IEEE CS Press, 1998.

[300] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th International Conference on Software Engineering*, pp. 41–50, ACM Press, 1995.

[301] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pp. 102–112, ACM, 2000.

[302] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *15th International Symposium on Software Reliability Engineering*, pp. 113–124, IEEE Computer Society, 2004.

[303] S. S. Yau and Z. Kishimoto, "A method for revalidating modified programs in the maintenance phase," in *Proceedings of International Computer Software and Applications Conference*, 1987.

[304] M. Harman, "Making the case for morto: Multi objective regression test optimization," in *ICST Workshops*, pp. 111–114, 2011.

[305] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[306] K. Deb and T. Goel, "Controlled elitist non-dominated sorting genetic algorithms for better convergence," in *Evolutionary Multi-Criterion Optimization*, vol. 1993 of *Lecture Notes in Computer Science*, pp. 67–81, Springer Berlin Heidelberg, 2001.

[307] D. Whitley, "The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best," in *Proceedings of the third international conference on Genetic algorithms*, (San Francisco, CA, USA), pp. 116–121, Morgan Kaufmann Publishers Inc., 1989.

[308] S. W. Mahfoud, "Niching methods for genetic algorithms," tech. rep., Illinois Genetic Algorithms Laboratory, 1995.

[309] G. Harik, "Finding multimodal solutions using restricted tournament selection," in *Proceedings of the Sixth International Conference on Genetic Algorithms*, (Pittsburgh, PA, USA), pp. 24–31, Morgan Kaufmann, 1995.

[310] H. Hemmati, A. Arcuri, and L. Briand, "Empirical investigation of the effects of test suite properties on similarity-based test case selection," in *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pp. 327–336, IEEE Computer Society, 2011.

[311] K. A. De Jong, *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[312] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, (Cambridge, MA, USA), pp. 41–49, Lawrence Erlbaum Associates, 1987.

[313] A. Della Cioppa, C. De Stefano, and A. Marcelli, "On the role of population size and niche radius in fitness sharing," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 6, pp. 580–592, 2004.

[314] A. Della Cioppa, C. De Stefano, and A. Marcelli, "Where are the niches? dynamic fitness sharing," *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 4, pp. 453–465, 2007.

[315] B. Sareni and L. Krähenbühl, "Fitness sharing and niching methods revisited," *IEEE Transactions on Evolutionary Computation*, vol. 2, no. 3, pp. 97–106, 1998.

[316] Y. Jie, N. Kharma, and P. Grogono, "Bi-objective multipopulation genetic algorithm for multimodal function optimization," *Evolutionary Computation, IEEE Transactions on*, vol. 14, no. 1, pp. 80–102, 2010.

[317] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 5, pp. 443–462, 2002.

[318] R. Battiti and G. Tecchiolli, "The reactive tabu search," *ORSA Journal on Computing*, vol. 6, no. 2, pp. 126–140, 1994.

[319] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms.* Wiley, 2001.

[320] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 6.1–6.42, 2013.

[321] M. Köppen and K. Yoshida, "Substitute distance assignments in nsga-ii for handling many-objective optimization problems," in *Evolutionary Multi-Criterion Optimization*, vol. 4403 of *Lecture Notes in Computer Science*, pp. 727–741, Springer Berlin Heidelberg, 2007.

[322] G. Strang, *Introduction to Linear Algebra.* Wellesley-Cambridge Press and SIAM, 4th ed., 2009.

[323] J. Smith and T. Fogarty, "Self adaptation of mutation rates in a steady state genetic algorithm," in *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pp. 318–323, 1996.

[324] A. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 3, no. 2, pp. 124–141, 1999.

[325] D. Thierens, "Adaptive mutation rate control schemes in genetic algorithms," in *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, vol. 1, pp. 980–985, 2002.

[326] N. Hansen, "The CMA evolution strategy: a comparing review," in *Towards a new evolutionary computation. Advances on estimation of distribution algorithms* (J. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea, eds.), pp. 75–102, Springer, 2006.

[327] N. Hansen and S. Kern, "Evaluating the CMA evolution strategy on multimodal test functions," in *Parallel Problem Solving from Nature PPSN VIII* (X. Yao *et al.*, eds.), vol. 3242 of *LNCS*, pp. 282–291, Springer, 2004.

[328] L. N. Trefethen and D. Bau III, *Numerical linear algebra.* Society for Industrial and Applied Mathematics, 1997.

[329] N. Holter, A. Maritan, M. Cieplak, N. Fedoroff, and J. Banavar, "Dynamic modeling of gene expression data," in *Proceedings of the National Academy of Sciences of United States of America*, vol. 98, pp. 1693–1698, 2001.

[330] J. A. Richards, *Remote Sensing Digital Image Analysis.* Springer-Verlag, 1993.

[331] T. Romo, J. Clarage, D. Sorensen, and G. N. J. Phillips, "Automatic identification of discrete substates in proteins: singular value decomposition analysis of time-averaged crystallographic refinements," *Proteins*, vol. 22, pp. 311–321, 1995.

[332] A. Auger and N. Hansen, "A restart cma evolution strategy with increasing population size," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 2, pp. 1769 – 1776, sept. 2005.

[333] K. Chellapilla, "Combining mutation operators in evolutionary programming," *IEEE Transactions on Evolutionary Computation*, vol. 2, no. 3, pp. 91 –96, 1998.

[334] Y.-W. Leung and Y. Wang, "An orthogonal genetic algorithm with quantization for global numerical optimization," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 1, pp. 41 –53, 2001.

[335] X. Yao, Y. Liu, and G. Lin, "Evolutionary programming made faster," *IEEE Transactions on Evolutionary Computation*, vol. 3, pp. 82–102, 1999.

[336] K. A. D. Jong and W. M. Spears, "An analysis of the interacting roles of population size and crossover in genetic algorithms," in *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, (London, UK, UK), pp. 38–47, Springer-Verlag, 1991.

[337] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evol. Comput.*, vol. 1, pp. 1–23, 1993.

[338] C. Fonseca and P. Fleming, "Multiobjective optimization and multiple constraint handling with evolutionary algorithms. i. a unified formulation," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 28, no. 1, pp. 26–37, 1998.

[339] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolution Computation*, vol. 8, no. 2, pp. 173–195, 2000.

[340] V. L. Huang, A. K. Qin, K. Deb, E. Zitzler, P. N. Suganthan, J. J. Liang, M. Preuss, and S. Huband, "Problem definitions for performance assessment of multi-objective optimization algorithm: Special session on constrained real-parameter optimization," tech. rep., Nanyang Technological University, Singapore, 2007.

[341] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable multi-objective optimization test problems," in *CEC 2002*, pp. 825–830, 2002.

[342] C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, 2001.

[343] S. G. E. Hyunsook Do and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.," *Empirical Software Engineering: An International Journal*, vol. 10, pp. 405–435, 2005.

[344] J. Cohen, *Statistical power analysis for the behavioral sciences.* Lawrence Earlbaum Associates, 2nd ed., 1988.

[345] H. Li, Y.-C. Jiao, L. Zhang, and Z.-W. Gu, "Genetic algorithm based on the orthogonal design for multidimensional knapsack problems," *Advances in Natural Computation*, vol. 4221, pp. 696–705, 2006.

[346] H. E. Aguirre and K. Tanaka, "Selection, drift, recombination, and mutation in multi-objective evolutionary algorithms on scalable mnk-landscapes," in *Evolutionary Multi-Criterion Optimization*, vol. 3410 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005.

[347] H. Maaranen, K. Miettinen, and A. Penttinen, "On initial populations of a genetic algorithm for continuous optimization problems," *Journal of Global Optimization*, vol. 37, pp. 405–436, Mar. 2007.

[348] Q. Zhang and Y.-W. Leung, "An orthogonal genetic algorithm for multimedia multicast routing," *Trans. Evol. Comp*, vol. 3, pp. 53–62, Apr. 1999.

[349] D. Jones and J. A. Eccleston, "Exchange and interchange procedures to search for optimum design," *Journal of the Royal Statistical Society*, pp. 238–243, 1980.

[350] D. R. Stinson, "Combinatorial designs: constructions and analysis," *SIGACT News*, vol. 39, no. 4, pp. 17–21, 2008.

[351] J. Zhu, G. Dai, and L. Mo, "A cluster-based orthogonal multi-objective genetic algorithm," *Computational Intelligence and Intelligent Systems*, vol. 51, pp. 45–55, 2009.

[352] E. Zitzler, D. Brockhoff, and L. Thiele, "The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration," in *Evolutionary Multi-Criterion Optimization*, vol. 4403 of *Lecture Notes in Computer Science*, pp. 862–876, Springer Berlin Heidelberg, 2007.