TESIS DOCTORAL

Heuristic Algorithms for the Optimization of Software Quality

Autor:

Javier Yuste Moure

Directores:

Abraham Duarte Muñoz

Eduardo García Pardo

Programa de Doctorado en Tecnologías de la Información y las Comunicaciones

Escuela Internacional de Doctorado

2024

This doctoral thesis has been partially supported by: grants PID2021-125709OA-C22, PID2021-126605NB-I00, funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe"; grant P2018/TCS-4566, funded by the Comunidad de Madrid and co-financed by the European Structural Funds ESF and FEDER; grant CIAICO/2021/224 funded by Generalitat Valenciana; grant M2988 funded by "Proyectos Impulso de la Universidad Rey Juan Carlos 2022"; and "Cátedra de Innovación y Digitalización Empresarial entre Universidad Rey Juan Carlos y Second Episode" (Ref. ID MCA06).

Copyright ©2024 Javier Yuste Moure. All rights reserved.

It is not enough to know, we must also apply;
it is not enough to will, we must also do.

Johann Wolfgang von Goethe

Acknowledgments

A doctoral thesis marks the end of a long path. Although it is authored by a single student, its existence would not be possible without the work of others. Standing on the shoulders of giants is a well-known metaphor to indicate that we build on previous discoveries to make intellectual progress. In this sense, this doctoral thesis is the product of truly excellent thinkers who have guided me through the fascinating world of scientific research.

Notwithstanding the inspiration that has come from many mentors over the years, I am especially grateful to Eduardo, whose professionalism, discipline, nonconformism, and analytical way of thinking incited me to pursue an academic career. Similarly, I must thank Abraham for giving me the opportunity to learn from such an outstanding researcher. Moreover, I am truly grateful to all my colleagues in both the research group and the department, who are a source of inspiration for me and many others.

Although the meaning of the aforementioned metaphor is well established, it can also be understood to include those who have worked to support us. In this sense, this doctoral thesis is also the product of family and friends, giants who inspire me every day. I would be nothing without my parents, whose sacrifices I will never be able to thank enough. I would be nothing without my brother, whose hard work is remarkable. I would be nothing without my friends, whose loyalty is unquestionable. And I would be nothing without Ana, whose love is unconditional. They are the ones who make it meaningful.

Abstract

Software quality is of utmost importance for the correct functioning of modern systems. The quality of software projects is measured by different attributes, such as efficiency, security, or understandability, among others. Without a proper design, the code becomes prone to errors and unsatisfactory. In this doctoral thesis, we study the optimization of software quality. In particular, we focus on the optimization of software maintainability, which is critical to the long-term success of software projects. The subject studied is known as the Software Module Clustering Problem, which is a well-known family of optimization problems in the area of Search-Based Software Engineering. We study four of these problems based on different quality metrics used to evaluate software systems. Two of them, Modularization Quality and Function of Complexity Balance, are studied as mono-objective problems. The other two problems, Maximizing Cluster Approach and Equal-size Cluster Approach, consider multiple quality metrics and are studied as multi-objective optimization problems. Given the complexity of these problems, which have been proven to be NP-complete, exact methods are impractical for the size of real-world software projects. Therefore, this doctoral thesis focuses on approximate methods. In particular, the use of three metaheuristic procedures is proposed: a Greedy-Randomized Adaptive Search Procedure combined with Variable Neighborhood Descent, a General Variable Neighborhood Search, and a Multi-Objective General Variable Neighborhood Search. To improve the efficiency of the aforementioned methods, several novel strategies are introduced, and an exhaustive study of neighborhood structures and their exploration is performed. Finally, the proposed methods have been validated by favorably comparing their performance with the best algorithms available in the related literature, on a dataset obtained from real software instances. The significance of the results obtained is supported by statistical tests.

Contents

Acknowledgments vii

Abstract ix

Contents xi

List of tables xv

List of figures xix

List of acronyms xxi

I PhD Dissertation 1

1 Introduction 3

1.1 Motivation . . . 4
1.2 Optimization . . . 7
1.2.1 Optimization problems . . . 8
1.2.2 Optimization methods . . . 10
1.3 Research methodology . . . 12
1.4 Hypothesis and objectives . . . 13

2 Problem definition 17

2.1 Modularization Quality . . . 20
2.1.1 BasicMQ . . . 20

2.1.2 TurboMQ . . . 22
2.2 Function of Complexity Balance . . . 23
2.3 Maximizing Cluster Approach . . . 26
2.4 Equal-size Cluster Approach . . . 27

3 Literature review 31
3.1 Large Neighborhood Search for the MQ problem . . . 38
3.2 Hybrid Genetic Algorithm for the FCB problem . . . 40
3.3 Two-Archive Artificial Bee Colony for the MCA and ECA problems . . . 42

4 Algorithmic proposal 45
4.1 Fundamentals of the algorithmic methodologies . . . 45
4.1.1 Greedy Randomized Adaptive Search Procedure . . . 46
4.1.2 Variable Neighborhood Search . . . 49
4.1.3 Multi-Objective Variable Neighborhood Search . . . 55
4.2 Algorithmic proposal for the MQ problem . . . 59
4.2.1 Preprocessing reduction phase . . . 60
4.2.2 Constructive phase . . . 61
4.2.3 Variable Neighborhood Descent . . . 64
4.3 Algorithmic proposal for the FCB problem . . . 66
4.3.1 Constructive procedure . . . 66
4.3.2 Variable Neighborhood Descent . . . 67
4.4 Algorithmic proposal for the MCA and ECA problems . . . 68
4.4.1 Constructive procedure . . . 69
4.4.2 Multi-Objective Shake . . . 72
4.4.3 Multi-Objective Variable Neighborhood Descent . . . 75
4.5 Neighborhood structures . . . 75
4.5.1 Neighborhood structures defined by operations that do not alter the number of modules . . . 76
4.5.2 Neighborhood structures defined by operations that increase the number of modules . . . 79

4.5.3 Neighborhood structures defined by operations that reduce the number of modules . . . 82
4.6 Advanced strategies . . . 86
4.6.1 Incremental evaluation of the objective functions . . . 86
4.6.2 Identification of promising areas in the search space . . . 90
4.6.3 Analysis of the contribution of the guiding functions . . . 95

5 Experiments **99**
5.1 Dataset . . . 99
5.2 Preliminary computational experiments . . . 102
5.2.1 GRASP-VND . . . 102
5.2.2 GVNS . . . 107
5.2.3 MO-GVNS . . . 114
5.3 Comparison with the state of the art . . . 124
5.3.1 Comparison of the GRASP-VND procedure with the best methods for the MQ problem . . . 124
5.3.2 Comparison of the GVNS procedure with the best methods for the FCB problem . . . 126
5.3.3 Comparison of the MO-GVNS procedure with the best methods for the MCA and ECA problems . . . 127

6 Conclusions and future work **131**
6.1 Conclusions . . . 131
6.2 Future work . . . 133
6.3 Contributions . . . 135

II Appendix **141**

A Dataset **143**

B Resumen en castellano **149**
B.1 Motivación . . . 150

B.2 Metodología . . . 152
B.3 Hipótesis y objetivos . . . 153
B.4 Definición del problema . . . 156
B.4.1 *Modularization Quality* . . . 158
B.4.2 *Function of Complexity Balance* . . . 161
B.4.3 *Maximizing Cluster Approach* . . . 162
B.4.4 *Equal-size Cluster Approach* . . . 165
B.5 Estado del arte . . . 165
B.6 Propuestas algorítmicas . . . 172
B.7 Resultados . . . 174
B.8 Conclusiones y trabajos futuros . . . 177
B.9 Contribuciones . . . 179

Bibliography **187**

Glossary **203**

List of tables

3.1 Classification of some families of problems identified in the SBSE research area according to the phase within a SDLC where they mostly arise. 32
3.2 Chronological summary of proposals for single-objective optimization in the SMCP family. 35
3.2 Chronological summary of proposals for single-objective optimization in the SMCP family. 36
3.3 Chronological summary of proposals for multi-objective optimization in the SMCP family. 37

4.1 Selection criteria used for each of the two vertices involved in a swap operation, for each of the proposed shake procedures. 73

5.1 Contribution of exploring each neighborhood to the quality of the initial solutions obtained by the constructive procedure within the GRASP-VND procedure 103
5.2 Comparison of the results obtained with the GRASP-VND procedure with different orderings of the proposed neighborhoods in the reduced dataset . . 104
5.3 Comparison of the performance of the GRASP-VND method proposed when implementing advanced strategies. 107
5.4 Contribution of exploring different neighborhoods in isolation within the GVNS procedure to the search process. 109
5.5 Comparison of the results obtained with the GVNS algorithm with different orderings of the proposed neighborhoods in the reduced dataset 110

5.6 Comparison of different neighborhoods explored within the shake component of the GVNS method . . . 111
5.7 Comparison of the results obtained by the GVNS method proposed when configuring the algorithm to stop after 5, 10, 15, or 20 iterations without improvement . . . 114
5.8 Comparison of the performance of the GVNS method when implementing different advanced strategies. . . . 115
5.9 Comparison of different shake procedures for the MCA problem. . . . 117
5.10 Comparison of different shake procedures for the ECA problem. . . . 117
5.11 Comparison of different values of k_{max} for the MCA problem. . . . 118
5.12 Comparison of different values of k_{max} for the ECA problem. . . . 118
5.13 Comparison of the results obtained with (Incremental) and without (Complete) using the incremental evaluation strategy for the MCA problem. . . . 119
5.14 Comparison of the results obtained with (Incremental) and without (Complete) using the incremental evaluation strategy for the ECA problem. . . . 120
5.15 Comparison of the results obtained with (Reduced) and without (Complete) reducing the size of the neighborhoods for the MCA problem. . . . 120
5.16 Comparison of the results obtained with (Reduced) and without (Complete) reducing the size of the neighborhoods for the ECA problem. . . . 121
5.17 Comparison of the average error rates obtained by removing some of the considered objectives in the MCA problem for the evaluation of the solutions. 122
5.18 Comparison of the average error rates obtained by removing some of the considered objectives in the ECA problem for the evaluation of the solutions. 122
5.19 Comparison of the results obtained by considering different sets of objectives as guiding functions for the MCA problem. . . . 123
5.20 Comparison of the results obtained by considering different sets of objectives as guiding functions for the ECA problem. . . . 124
5.21 Comparison of the results obtained by the proposed GRASP-VND procedure and the state-of-the-art LNS method for the MQ problem. . . . 126
5.22 Comparison of the results obtained with the method proposed in this research, GVNS, and the best known algorithm, a HGA, for the FCB problem 128

5.23 Comparison of the proposed MO-GVNS with several state-of-the-art methods for the MCA problem. 129
5.24 Comparison of the proposed MO-GVNS with several state-of-the-art methods for the ECA problem. 129

A.1 Small instances contained in the dataset. These instances have a number of vertices between 2 and 68. 143
A.2 Medium instances contained in the dataset. These instances have a number of vertices between 74 and 182. 146
A.3 Large instances contained in the dataset. These instances have a number of vertices between 190 and 377. 147
A.4 Very large instances contained in the dataset. These instances have a number of vertices between 413 and 1161. 147

B.1 Resumen cronológico de propuestas para problemas monoobjetivo pertenecientes al SMCP. 169
B.1 Resumen cronológico de propuestas para problemas monoobjetivo pertenecientes al SMCP. 170
B.2 Resumen cronológico de propuestas para problemas multiobjetivo pertenecientes al SMCP. 171
B.3 Comparación de los resultados obtenidos con el método GRASP-VND propuesto y el método LNS del estado del arte para el problema MQ. 175
B.4 Comparación de los resultados obtenidos con el método GVNS propuesto y el método HGA del estado del arte para el problema FCB. 176
B.5 Comparación de los resultados obtenidos con el método MO-GVNS propuesto y los métodos de referencia del estado del arte para el MCA. 176
B.6 Comparación de los resultados obtenidos con el método MO-GVNS propuesto y los métodos de referencia del estado del arte para el problema ECA. 176

List of figures

1.1 Research methodology. 14

2.1 Graphical representation of dependencies of a software project in an MDG. 18
2.2 Graphical representation of an example MDG for a software project. 19
2.3 Graphical representation of an example MDG for a software project (2). . . 19
2.4 Graphical representation of a trivial solution for a software project. 20
2.5 Calculation of the quality of a solution in the MQ problem. 24
2.6 Calculation of the quality of a solution in the FCB problem. 25
2.7 Calculation of the quality of a solution in the MCA problem. 28
2.8 Calculation of the quality of a solution in the ECA problem. 29

4.1 Restricted Candidate List in the GRASP constructive procedure. 49
4.2 Illustration of VNS ideas. 52
4.3 Preprocessing phase of the algorithmic proposal for the MQ problem. . . . 62
4.4 Agglomerate constructive phase of the proposed MO-GVNS method. . . . 71
4.5 Representation of a solution generated by the MO-GVNS constructive procedure in the objective space. 72
4.6 Insert operation. 77
4.7 Swap operation. 78
4.8 Extract operation. 80
4.9 Split operation. 81
4.10 Destroy operation. 83
4.11 Merge operation. 85
4.12 Incremental evaluation of the TurboMQ value of a solution x. 89

4.13 Incremental evaluation of the FCB value of a solution x (I). 90
4.14 Incremental evaluation of the FCB value of a solution x (II). 91
4.15 Identification of promising insert operations for a given solution. 93

5.1 Number of vertices and edges of each instance in the dataset. 101
5.2 Number of vertices and density of each instance in the dataset. 101
5.3 Stopping criterion analysis for the GRASP-VND method proposed 106
5.4 Comparison of the average quality of the best solution found for the instances in the reduced dataset over time with different values of k_{max} in the GVNS method proposed. 113

6.1 Timeline of the most relevant events associated with this doctoral thesis. . . 136

B.1 Diagrama de actividad de la metodología seguida en esta tesis doctoral. . . 154
B.2 Representación gráfica de la estructura de un proyecto *software* en un MDG. 157
B.3 Representación gráfica de una posible solución para un proyecto *software* ficticio. 158
B.4 Cálculo del valor de MQ para una solución. 160
B.5 Cálculo del valor de FCB para una solución. 162
B.6 Cálculo de los valores de los objetivos del problema MCA para una solución. 164
B.7 Cálculo de los valores de los objetivos del problema ECA para una solución. 166
B.8 Cronología de los eventos más relevantes relacionados con esta tesis doctoral. 181

List of acronyms

ABC	Artificial Bee Colony
BVNS	Basic Variable Neighborhood Search
C	Coverage
CA	Convergence Archive
CCCS	Cohesion, Coupling, package Count index, and package Size index
DA	Diversity Archive
DQCQ	Dependency Quality and Connection Quality
E-CDGM	Evolutionary Call-Dependency Graph Modularization
E-ECA	Extended Equal-size Cluster Approach
E-MCA	Extended Maximizing Cluster Approach
ECA	Equal-size Cluster Approach
EoD	Estimation of Distribution
EOF	Entropy-based Objective Function
FA	Firefly Algorithm
FCB	Function of Complexity Balance
GA	Genetic Algorithm
GAHC	Genetic Algorithm with Hill Climbing
GLMPSO	Grid-based Large-scale Many-objective Particle Swarm Optimization
GMA	Graph-based Modularization Algorithm
GRAFO	Group for Research in Algorithms For Optimization
GRASP	Greedy Randomized Adaptive Search Procedure
GVNS	General Variable Neighborhood Search
HC	Hill Climbing
HGA	Hybrid Genetic Algorithm
HS	Harmony Search
HV	Hypervolume

IBEA	Indicator-Based Evolutionary Algorithm
IEC	Interactive Evolutionary Computation
IFF	Interactive Fitness Function
IGD	Inverted Generational Distance
IGD+	Inverted Generational Distance plus
ILS	Iterative Local Search
ISO	International Organization for Standardization
JCR	Journal Citation Reports
LCC	Linear Compound Criteria
LNCS	Lecture Notes in Computer Science
LNS	Large Neighborhood Search
MaABC	Many-objective Artificial Bee Colony
MAEA	Multi-Agent Evolutionary Algorithm
MAEB	Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados
MCA	Maximizing Cluster Approach
MDG	Module Dependency Graph
MFMC	Multi-Factor Module Clustering
MHypEA	Multi-objective Hyper-heuristic Evolutionary Algorithm
MIC	Metaheuristics International Conference
MNCC	MQ, Non-extreme distribution, Coupling, and Cohesion
MO-GVNS	Multi-Objective General Variable Neighborhood Search
MO-VND	Multi-Objective Variable Neighborhood Descent
MO-VNS	Multi-Objective Variable Neighborhood Search
MOEA/D	Multi-Objective Evolutionary Algorithm based on Decomposition
MOF	Multi-Objective Fitness function
MOP	Multi-objective Optimization Problem
MQ	Modularization Quality
MS	Modularization Quality measure based on similarity
NAHC	Next Ascent Hill Climbing
NSGA-II	Non-dominated Sorting Genetic Algorithm II
NSGA-III	Non-dominated Sorting Genetic Algorithm III
PESA2	Modified Pareto Envelop-Based Selection Algorithm
PSO	Particle Swarm Optimization

RVNS	Reduced Variable Neighborhood Search
SA	Simulated Annealing
SAHC	Steepest Ascent Hill Climbing
SBSE	Search-Based Software Engineering
SDLC	Software Development Life-Cycle
SE	Software Engineering
SJR	Scientific Journal Rankings
SMCP	Software Module Clustering Problem
SSBSE	Symposium on Search Based Software Engineering
SSH	Structure of packages, Semantics coherence, and History of changes
TA-ABC	Two-Archive Artificial Bee Colony
TAA	Two-Archive Algorithm
URJC	Universidad Rey Juan Carlos
VLSN	Very Large Scale Neighborhood search
VND	Variable Neighborhood Descent
VNS	Variable Neighborhood Search

Part I

PhD Dissertation

Chapter 1

Introduction

Software is of utmost importance for the correct functioning of modern systems nowadays. It is used to control any electronic device, from widespread tools such as mobile phones to critical aircraft systems or medical instruments. Computer programming has revolutionized the world in the last decades and presumably will continue to do so in the following years. As software systems become larger and more complex, navigating them becomes increasingly difficult. The more sophisticated the system, the harder it is to understand it. Moreover, software systems are usually subject to constant change in order to adapt to new user needs, add features, correct defects, or improve performance. For these reasons, software systems often deteriorate over time, becoming prone to errors. According to recent studies, low-quality software was responsible for costs of up to 2.08 trillion dollars in 2020, considering only the United States of America [77]. In addition, software errors can critically impact the outcomes of a system. For example, the launch of satellite Ariane 5 in 1996 [135], the failed landing of Mars Polar Lander in 1999 [5], or the error in Starliner in 2019 [92], are among the most notorious and costly failures caused by software errors.

In an attempt to alleviate some of the problems mentioned above, there has been an increase in the effort made by the scientific community to automatically improve and correct software systems. Search-Based Software Engineering (SBSE) is a research area that tackles Software Engineering (SE) tasks as optimization problems. By applying optimization techniques to SE endeavors, the goal of the SBSE community is to improve the quality of software systems, correct code errors, and, in general, facilitate the work of software

developers.

In this doctoral thesis, we study the Software Module Clustering Problem (SMCP), a family of problems that can be found within the SBSE research community. In particular, we study four different problems of the family and propose several heuristic algorithms to tackle them. For the algorithms proposed, we design some advanced strategies that help improve their performance and efficiency. To empirically validate our findings, we favorably compare the results obtained by the proposed methods with the most recent state-of-the-art algorithms in each variant of the problem studied.

In this chapter, we first discuss the motivation to tackle the SMCP in Section 1.1. Then, we present an introduction to optimization in Section 1.2. Next, we describe the methodology followed during this research in Section 1.3. Finally, we conclude this chapter by presenting the hypothesis and objectives of this doctoral thesis in Section 1.4.

1.1 Motivation

With the widespread use of software programs and electronic devices, the costs of software development and maintenance have increased significantly in recent years [61]. Moreover, software development is a complex endeavor. According to a report that surveyed software projects between 2011 and 2015, 56% of the projects were over budget, 60% were delivered late, and 44% did not contain the expected set of features [1]. Furthermore, only 56% of the software projects surveyed delivered customer and user satisfaction. These results are particularly worrying for the largest software projects, which had a success rate of only 6% to 11%. As stated in the report, "*complexity is one of the main reasons for project failure*" [1].

The life cycle of a software project contains all the activities performed to evolve a system from its conception until its retirement [61]. In order to ensure that high-quality code is created, it is advisable for software engineers to follow a structured approach known as a Software Development Life-Cycle (SDLC). An SDLC is a framework that involves the definition, ordering, and transition criteria of different stages that contain several processes and activities related to the life cycle of the system [61]. Although the exact number of phases can be argued, every SDLC contains at least a requirements definition, analysis and

design, software development, testing, and maintenance phases. Implementing a defined SDLC model has been shown to improve the quality of software products [139]. A great example is NASA[1], which, after performing software process improvements for four years, reduced the error rate for on-board software in their space shuttle from 2 errors per 1000 lines of code to only 0.11 errors per 1000 lines of code (a reduction of 94%) [58].

In the SDLC, there are two major phases: software development and software maintenance. In the software development phase, a system element is produced. That is, the software development phase transforms requirements into actions that create a system element [61]. Software maintenance, on the other hand, focuses on providing cost-effective support to the system, sustaining its capability to provide a service [61]. This support includes different corrective, adaptive, perfective, and preventive actions [13]. Software maintenance is often regarded as a somewhat less important phase, resulting in a lack of recognition, motivation, and support [34, 82]. However, maintenance is the most costly phase of the SDLC. Studies have shown that up to 80% of the total costs are spent on software maintenance [22]. Interestingly, most of the effort in this phase is devoted to understanding the existing software [103]. Indeed, the need for software maintenance and the increasing complexity of software projects throughout their life cycle were already explained by Lehman's laws of software evolution, formulated between 1974 and 1996 [81]. Lehman distinguishes programs in three categories: S, P and E [80]. The Lehman's laws of sofware evolution apply to E-programs, which are programs written to perform some real-world activity, as opposed to S- and P-programs which are written to perform specific tasks and thus do not evolve over time. Among the eight laws proposed by Lehman, four of them are directly related to the importance and need for software maintenance:

- Continuing change: a "*program that is used must be continually adapted else it becomes progressively less satisfactory*" [81].
- Continuing growth: "*functional content of a program must be continually increased to maintain user satisfaction over its lifetime*" [81].
- Increasing complexity: "*as a program is evolved its complexity increases unless work*

[1] https://www.nasa.gov/

is done to maintain or reduce it" [81].

- Declining quality: "*programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment*" [81].

As stated by the aforementioned laws, the continuous change and growth of software systems result in an increasing complexity and a deterioration of software quality. According to the International Organization for Standardization (ISO)[2], product quality is defined by eight attributes: functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability [64]. Amid these attributes, maintainability is defined as the "*degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers*", including corrections, improvements, or adaptations [64]. Moreover, five characteristics of maintainability, which are tightly intertwined, are defined: modularity, reusability, analysability, modifiability, and testability. Since maintainability and understandability affect the ease with which a software system can be modified over time, they are important aspects for the long-term success of software projects, and ignoring them early can contribute to considerably more effort [42].

Due to the fact that the main problems impacting software maintainability are "*software comprehension problems*", one of the keys to achieve a high level of maintainability is dealing with the static structure of software systems [23]. Organizing source code into different components facilitates the comprehension of each component independently. However, using only components to represent a system is not sufficient to gain robust, maintainable, and reusable designs. Instead, a higher level of abstraction that involves packages or modules (groups of components) is needed for large codebases to facilitate the comprehension of each module independently [6]. In this context, modularity is defined as the set of "*software attributes that provide a structure of highly independent components*" [63]. In a modular organization, software components within the same module should be highly related (high cohesion), while components within different modules should be weakly connected (low coupling). Indeed, there exist several studies that positively correlate modularity with maintainability [4, 12, 19]. There are different notions of what a software component is,

[2] https://www.iso.org/home.html

such as a file, a class, a package, etc. Although in some contexts, the words "module" and "component" are synonymous, it is worth mentioning that the definition of these two terms is not standardized [63]. Here, we will utilize the word "component" for individual elements (files and classes) and the word "module" for collections of components (packages or folders).

Due to the relations between software modularity and maintainability with software quality and costs, it is of paramount importance for the long-term success of software projects to organize their components in a highly modular structure. Such a structure facilitates the comprehension of the systems, and thus affects positively the quality of software projects. In particular, a good modular structure directly benefits all the characteristics of software maintainability: modularity, reusability, analyzability, modifiability, and testability. The SMCP is a family of optimization problems that focuses on the optimization of software modularity to reduce the maintenance costs of software projects and improve their quality. The main motivation for this doctoral thesis lies in the design and implementation of algorithmic approaches to improve the results obtained for the SMCP.

1.2 Optimization

Optimization, the search for the best solution for a given problem, is a process inherent to any form of life. Darwin's theory of evolution is, in essence, the description of an optimization process: the maximization of adaptability to the environment [46]. However, the field of optimization research has not received much attention until the past century [46]. Although there exist known works on optimization prior to 1947, this date marked the beginning of a new period on optimization with the proposal of the simplex method [29]. Now, optimization, defined as the "*search process that seeks for the best solution among the set of all possible solutions for a given problem*" [38], has become an important research field that integrates efforts from mathematics, computer science, management science, and artificial intelligence, among others. In this section, we introduce and define some important concepts of optimization research that are important for the context of this doctoral thesis. In particular, in Section 1.2.1, we describe optimization problems from a formal point of view. Then, in Section 1.2.2, we describe approaches to solve optimization problems.

1.2.1 Optimization problems

Typically, optimization problems have three fundamental elements. First, a method is needed to compare solutions. This method is commonly known as the objective function, and it assigns a numerical value to any possible solution to the problem. Second, a set of decision variables must be defined. Each decision variable can hold a value in a given range. A particular combination of values for the set of decision variables represents a solution to the problem. The set of all solutions that can be represented by the combination of values that the decision variables can hold represents the search space of the problem. Finally, a set of constraints might exist that limit the possible values or combinations of the values of the aforementioned variables. The set of solutions that satisfy the set of constraints is commonly known as the set of feasible solutions.

From a formal point of view, an optimization problem P can be defined as a 3-tuple $P = (f, S, FS)$, such that:

$$P = \begin{cases} Opt. : f(x) & \text{objective function} \\ s.t., & \\ x \in FS \subset S & \text{feasible (FS) and possible (S) solutions} \end{cases} \tag{1.1}$$

where f is the objective function, FS is the set of feasible solutions, and S is the set of possible solutions. The goal then is to find the solution $x^\star$ among the set of feasible solutions that either maximizes or minimizes the objective function, depending on the particular problem at hand. For the sake of clarity and without loss of generality, for the remainder of this section, we assume that every objective function must be maximized. Therefore, the goal is to find the solution $x^\star$ among the set of feasible solutions that maximizes the value of the objetive function:

$$x^\star = \underset{x \in FS}{\arg\max}(f(x)). \tag{1.2}$$

In real life, there are countless problems that can be modeled and tackled as optimization problems. Depending on the values that the variables for these problems can have, they are generally divided into two main categories: continuous and discrete optimization [38].

In continuous optimization, the variables are continuous quantities, real numbers. Therefore, the number of possible solutions is infinite. On the other hand, in discrete optimization problems, the variables are not continuous but discrete. Thus, the set of solutions is finite, although it may be very large. Many real-life problems need to be defined with discrete variables, since the resources are indivisible (e.g., machines, people, etc.) [134]. Within the family of discrete optimization problems, a particular type of problems can be found: combinatorial optimization problems. In these problems, there exist some discrete variables and a finite search space. The solution is typically a set of natural numbers that might be represented as a permutation, a graph, or another structure. Many real-life optimization problems are combinatorial, such as routing [79, 115], scheduling [113], social network analysis [84, 114], logistic [109], and graph embedding problems [21], among others.

The distinction between continuous and discrete optimization problems is based on the nature of the decision variables that represent the set of possible solutions to the problem at hand. However, optimization problems can also be distinguished based on the number of objective functions to be optimized. In the aforementioned definition of an optimization problem, only one objective function was considered. There, comparing the quality of different solutions is trivial. However, for many problems, there does not exist a unique function to optimize, but rather a set of different and conflicting objective functions. These are called Multi-objective Optimization Problems (MOPs).

Formally, an MOP is defined as:

$$\text{MOP} = \begin{cases} Opt.: F(x) = (f_1(x), f_2(x), ..., f_n(x)) & \text{vector of objectives} \\ s.t., & \\ x \in FS \subset S & \text{feasible (FS) and possible (S) solutions} \end{cases} \tag{1.3}$$

where $F(x)$ is the vector of objectives to be optimized. This vector, which represents the quality of the solution, can be represented in a n-dimensional space, called the objective space.

In MOPs, the objective functions are in conflict with each other. (In practice, if the objective functions are not conflicting, then the problem can be tackled as a single-objective

problem.) This means that improving one objective usually results in the deterioration of others. This situation makes it difficult to compare solutions. Given two solutions x and x' that have two different objectives such that $f_1(x) > f_1(x')$ and $f_2(x) < f_2(x')$, it is not clear which solution is better. In MOPs, a dominance relation is defined to compare the quality of solutions. This concept of dominance has its roots in the work of Edgeworth and Pareto [40, 111]. An objective vector $F(x) = (v_1, v_2, ..., v_n)$ is said to dominate another objective vector $F(x') = (v'_1, v'_2, ..., v'_n)$ if and only if every component in $F(x)$ is greater than or equal to the corresponding component in $F(x')$ and at least one component in $F(x)$ is strictly greater than the corresponding component in $F(x')$. Formally,

$$\forall i \in \{1, ..., n\} : v_i \geq v'_i \wedge \exists i \in \{1, ..., n\} : v_i > v'_i.$$

Given the aforementioned definition of dominance, a Pareto optimal solution can be defined as a solution in which it is impossible to improve any objective without deteriorating at least another one. (There exist other concepts of dominance, such as weak, strong and ε-dominance, which are not discussed here [134].) Then, the objective of MOPs is to find the Pareto optimal front, a set of Pareto optimal solutions. In practice, however, the Pareto optimal front is usually unknown and it is not possible to certify that a given Pareto front is optimal. Then, it is normally sufficient to find an approximate Pareto front. That is, a set of good solutions that are not dominated by the rest of the solutions in the same set.

In MOPs, due to the dominance relation, the number of Pareto optimal solutions is directly related to the number of objectives studied. At least all optimal solutions considering each objective in isolation are Pareto optimal solutions when the objectives are considered altogether in an MOP. Moreover, since there is not a clear numerical criterion for comparing solutions, the final choice depends on a decision maker, which adds another layer of complexity to MOPs.

1.2.2 Optimization methods

Regardless of the type of optimization problem being studied, there exist different approaches to find the best possible solutions. For combinatorial optimization problems, it is always possible to find the best solution and certify that it is indeed the best one, simply

by evaluating all possible solutions. The family of algorithms that are designed to find the optimal solution and certify its optimality is known as exact algorithms. However, in reality, it is often impractical to apply exact algorithms due to the vast number of possible solutions that usually exist for problems of interest. For this reason, it is frequently preferred to find a high-quality solution in a short or acceptable computing time, even if it is not certified as optimal. This is the goal of approximate algorithms.

Approximate algorithms try to find a high-quality solution in short computing times by applying some intelligence or heuristics in the search process. The term heuristic comes from the greek term *heuriskein*, which means "to find" or "discover"[3]. The meaning, however, has changed over time. The modern meaning of the term "heuristic" was first coined by G. Polya in 1957 [118]. In optimization, heuristics are defined as "simple procedures, often guided by common sense, that are meant to provide good but not necessarily optimal solutions to difficult problems, easily and quickly" [146]. By applying some strategies, heuristics explore only a subset of promising solutions among the whole set of possible solutions. Therefore, heuristics are approximate in the sense that they do not guarantee optimality.

Typically, heuristics can be classified in constructive and local search procedures. Constructive procedures start with an empty or incomplete solution and perform several operations to obtain a complete solution, usually feasible. Generally, the purpose of constructive procedures is to generate an initial solution for a later improvement procedure. On the other hand, a local search procedure starts from a complete solution and iteratively performs moves or operations on the solution to obtain a better one. At each step, the set of solutions that can be reached by applying a particular operation is denoted as a neighborhood. Therefore, a local search is the process of iteratively selecting a solution within the neighborhood of the current solution. Operations usually involve adding or dropping problem-specific components from the solution, or exchanging the position of two different elements.

Although heuristics are great at finding good solutions in a short computing time, they suffer from a major drawback: their inability to continue the search upon becoming trapped

[3] The term "Eureka!", famously known due to the popular story of Archimedes, means "I have found it!".

in local optima [46]. Broadly speaking, a local optimum is a solution that is better than every solution in its surrounding search space. A basin of attraction is a region of the search space where any gradient descent method leads to a local optimum. If the objective function is convex, then there exists only one local optimum, which is the global optimum. On the other hand, if the objective function is non-convex, then there may exist several local optima. Once trapped in a local optimum, simple heuristics are usually unable to escape the basin of attraction and continue exploring other regions of the search space.

The aforementioned problem has led researchers to develop metaheuristic approaches. A metaheuristic is a strategy that guides some underlying heuristics "*to create a process capable of escaping from local optima and performing a robust search of a solution space*" [50]. Metaheuristics are high-level, problem-independent procedures that provide some guidelines to develop heuristic algorithms, with the aim of overcoming local optima. They combine intensification (extensively exploring a promising region of the search space) and diversification (exploring different regions of the search space to identify promising ones) strategies to perform a more efficient search. Given their suitability to address real problems of practical interest and their ability to escape local optima, metaheuristic procedures have been the subject of extensive research in the field of optimization in the last few decades [38, 50, 91, 134].

1.3 Research methodology

The first recorded use of the term "research" dates from 1577, with the meaning of "careful or diligent search" [2]. Although there is no uniquely accepted all-encompassing definition of research, a generally accepted modern definition is "creative and systematic work undertaken to increase the stock of knowledge [...] and to devise new applications of available knowledge" [106]. Therefore, research must be performed following a systematic methodology. With respect to the scientific method, there is no uniquely accepted way of performing formal research. However, there exist some steps that are commonly accepted to be part of any research process: observation, hypothesis, experimentation, analysis of data, and conclusion. First, an observation or a set of observations is made. Then, a hypothesis is formulated that explains the observed behavior, and a set of experiments is designed to test

the hypothesis. Once the experiments have been performed, the data obtained are analyzed and some conclusions are drawn. This process is usually iterative, which means that data analysis often results in new observations and/or hypotheses.

Although the general overview of the scientific process is well known, the details differ depending on the area of study. In the field of heuristic research, which is an experimental area of study, the scientific process can be extended as illustrated in Figure 1.1. As can be seen in the activity diagram, the process is made up of ten steps. First, the problem is identified and its characteristics are examined (step 1). Then, the state of the art is studied (step 2). As a result, the most relevant algorithms are recognized, and the most appropriate set of reference instances used for comparison purposes is determined. Once these steps have been performed, a hypothesis is made considering the previous observations (step 3). Next, an algorithmic procedure is designed to test the hypothesis (step 4). This algorithm is then implemented to experimentally test the aforementioned hypothesis (step 5). Usually, computational experiments with algorithms are performed to accomplish one of two objectives: (i) to analyze the performance of an algorithm in isolation or (ii) to compare the performance of different algorithms for the same problem [14]. The first is performed in step 6, while the latter is performed in step 8. In general, algorithms should be tested against the best available methods for the problem being studied. In addition, well-known heuristics, even if not state-of-the-art for the problem at hand, provide valuable points of reference. As can be observed, after each experimentation phase, there exists a data analysis phase, where the obtained results are analyzed (steps 7 and 9). These analyses often result in a modification of the initial hypothesis and further iterations of the algorithmic design. Finally, a research article is published to share the relevant findings with the scientific community (step 10). Moreover, given the experimental nature of the field, the research article should contain enough details to allow the reproducibility of the experiments.

1.4 Hypothesis and objectives

In this doctoral thesis, the main goal is to design and implement optimization algorithms for the SMCP that outperform current state-of-the-art methods and can help to improve the maintainability of software projects. The hypothesis of this doctoral thesis is enunciated as

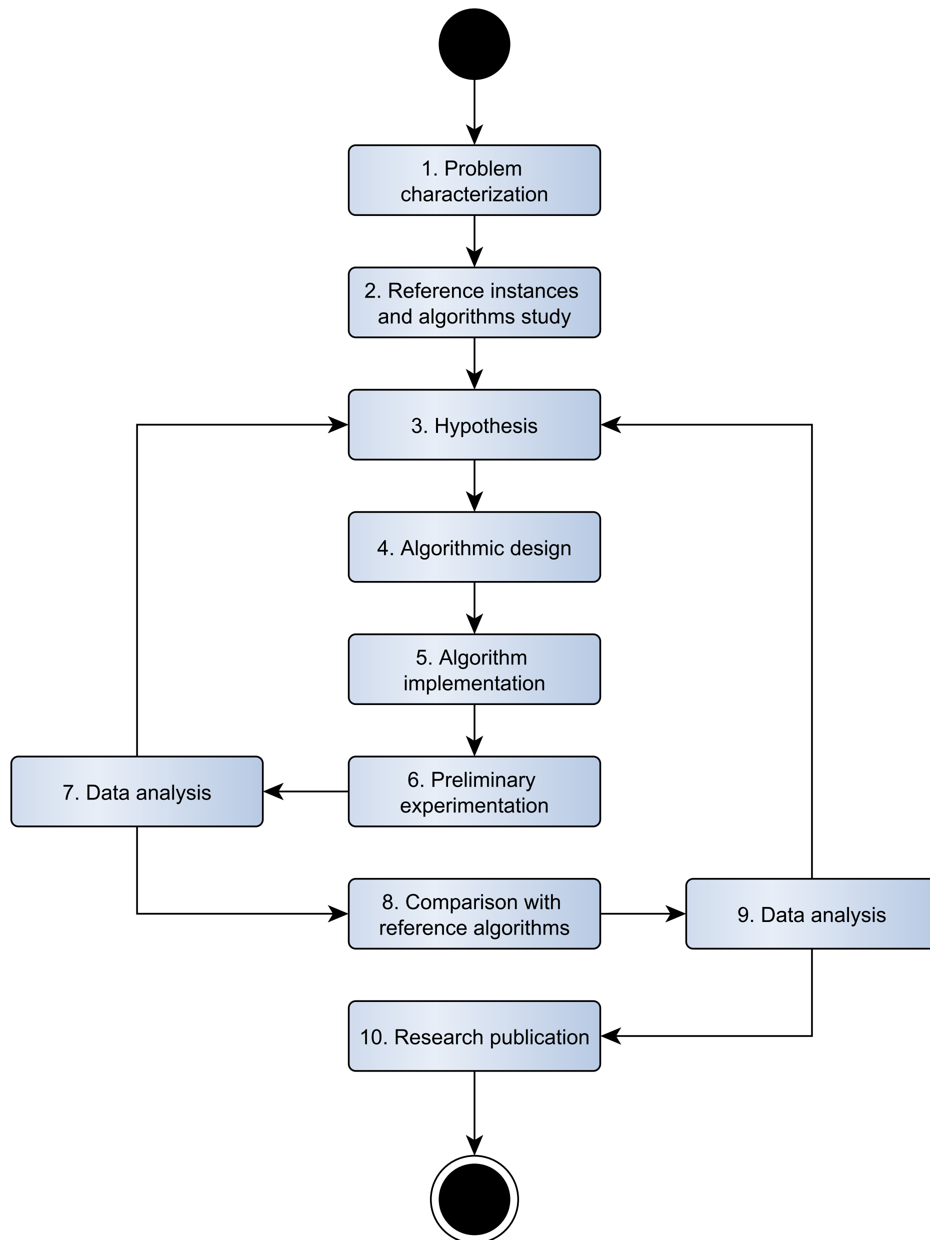


Figure 1.1 UML activity diagram [62] illustrating the research methodology followed in the field of heuristic research.

follows:

> "*The structure of software projects can be improved by modeling software maintainability as an optimization problem and implementing trajectory-based metaheuristics, which can obtain solutions of higher quality than population-based metaheuristics, currently present in the state of the art by leveraging domain-specific knowledge of the problem at hand to implement advanced strategies*".

To achieve the aforementioned goal and test the hypothesis, several partial objectives are considered. In particular:

1. **Review the literature of the problem**. The literature about software modularization and the SMCP must be studied in order to: (i) understand the problem and the motivation behind it; (ii) identify the current gaps in the literature and promising lines of research; (iii) study the objective functions and variants of the problem that exist in the literature, their differences, and advantages; and (iv) identify the reference algorithms and instances for the problem.

2. **Study the modeling of the problem**. A thoughtful analysis of the characteristics of the problem and the representation of the structure of software systems will lead to the design of efficient algorithms that are particularly suitable for the specifics of the problem.

3. **Collect a relevant dataset of instances**. In order to experimentally validate the findings and compare the performance of different algorithms, it is important to use a reference dataset curated by the community and accepted in the field.

4. **Implement the reference state-of-the-art algorithms**. In order to experimentally validate the algorithmic proposals, they must be compared with the best algorithms available for the problem at hand. To perform a fair comparison, all algorithms being compared should be executed in a uniform computing environment. Therefore, the algorithms should be implemented to be executed under the same conditions.

5. **Design and develop optimization algorithms based on trajectory-based metaheuristic frameworks**. These optimization algorithms will be designed with the

characteristics of the problem and its variants in mind.

6. **Parameterize the methods designed**. A trade-off in algorithmic design should be made between specificity for the problem at hand and generality for a wide variety of problems. Typically, heuristic algorithms include some parameters that can be tuned to adapt their behavior. These parameters, if any, should be tuned prior to comparison with other algorithms using a set of instances different from the one used for the comparison.

7. **Analyze the behavior of the proposed algorithms**. The behavior of the algorithm should be analyzed to study its strengths and weaknesses. The speed of convergence, the ability to escape from local optima, the robustness or the contribution of each component within the algorithm are among the set of characteristics that should be studied.

8. **Compare the proposed algorithms with the best methods in the state of the art**. This comparison should use realistic and relevant instances, be unbiased, generate valuable data, and be reproducible. Furthermore, to ensure fairness, a uniform computing environment should be used.

9. **Analyze the results obtained**. The data obtained from the experimentation should be analyzed and validated in order to test the hypotheses. This analysis can lead to further experiments or improvements in the design of the algorithm.

10. **Document the process and draw conclusions**. The research process should be documented in detail to share the relevant findings. The resulting document should be unbiased and allow for the reproducibility of the experiments. In addition, efforts should be made to draw conclusions that advance the available knowledge of either the problem studied or the algorithmic strategies designed.

11. **Submit the findings to peer review processes**. The documented research process should be submitted to relevant conferences and scientific journals for peer review and, if acceptable, be published to share the findings with the scientific community.

Chapter 2

Problem definition

In this doctoral thesis, we study the SMCP. In these problems, software projects are commonly represented in a graph structure known as an Module Dependency Graph (MDG) [60]. An MDG is a type of Artifact Dependency Graph, which represents the dependencies between components in a graph structure. There exist complementary types of graph to model other information about software projects that are not discussed here [60]. More formally, an MDG is a directed weighted graph $G = (V, E, W)$, where V is the set of vertices, E is the set of edges, and W is the set of weights associated with the edges in E. In this context, the vertices represent the components of the source code, the edges represent the dependencies between those components, and the weights represent the strength of the dependencies. In Figure 2.1, we show the MDG of a fictitious software project. This project has ten different components. Component one, represented by $v1$, has two dependencies (e.g., method calls, inheritance, etc.) toward component two, represented by $v2$. Therefore, the edge that connects $v1$ with $v2$ has an associated weight of two. Similarly, component four ($v4$) depends on components one ($v1$), three ($v3$), six ($v6$), and seven ($v7$), etc.

Given an MDG, a solution for the SMCP is represented by a clustering of the vertices of the graph. That is, a solution is a set $M = \{m_1, m_2, ..., m_n\}$ of non-empty disjoint subsets of vertices in V, where n represents the number of modules or clusters and $1 \leq n \leq |V|$. A solution M is called trivial if $n = 1$ or $n = |V|$. In Figure 2.2 and Figure 2.3, we illustrate two possible solutions for the aforementioned MDG. As can be observed, in the

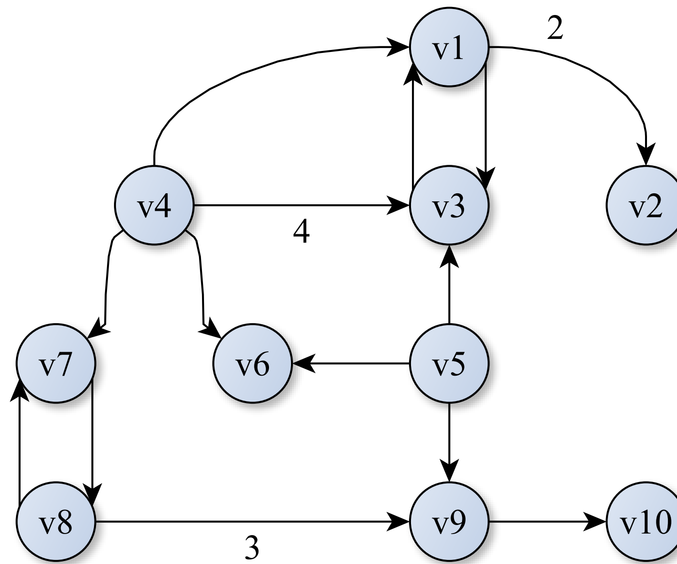


Figure 2.1 An example of an MDG of a software project. Vertices represent the components of the source code, edges represent the dependencies between those components, and weights represent the strength of the dependencies. For the sake of simplicity, only weights with a value greater than one are depicted.

first solution, the components have been grouped into three different modules, m_1, m_2, and m_3. In particular, $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$. In the second solution, the components have been grouped into two different modules, $m_1 = \{v1, v2, v3, v4\}$ and $m_2 = \{v5, v6, v7, v8, v9, v10\}$. Finally, Figure 2.4 represents a trivial solution for the MDG, since there is only one module ($n = 1$).

In the SMCP, there exist several problem variants that differ on how to evaluate the quality of modular organizations. In the following sections, the most relevant variants studied in this doctoral thesis are discussed. In particular, Modularization Quality (MQ) is presented in Section 2.1, Function of Complexity Balance (FCB) is presented in Section 2.2, Maximizing Cluster Approach (MCA) is presented in Section 2.3, and Equal-size Cluster Approach (ECA) is presented in Section 2.4.

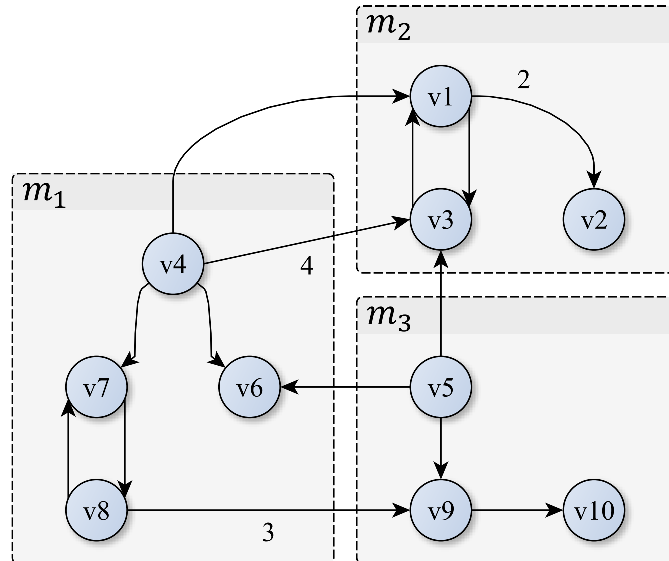


Figure 2.2 A modularized MDG of the software project represented in Figure 2.2 with three modules: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$.

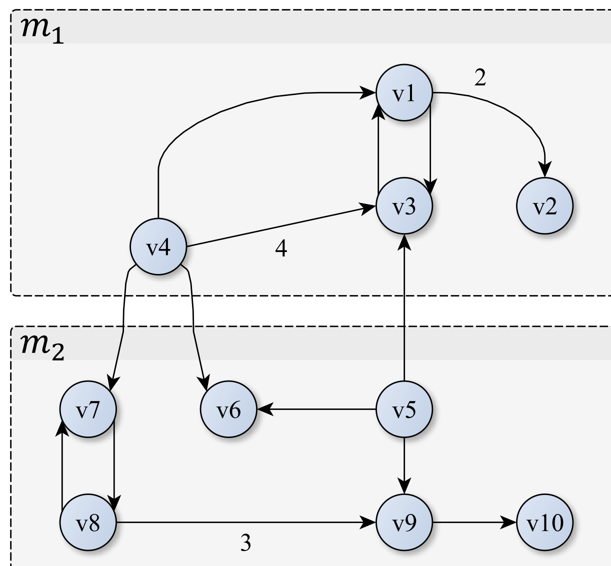


Figure 2.3 A modularized MDG of the software project represented in Figure 2.2 with two modules: $m_1 = \{v1, v2, v3, v4\}$ and $m_2 = \{v5, v6, v7, v8, v9, v10\}$.

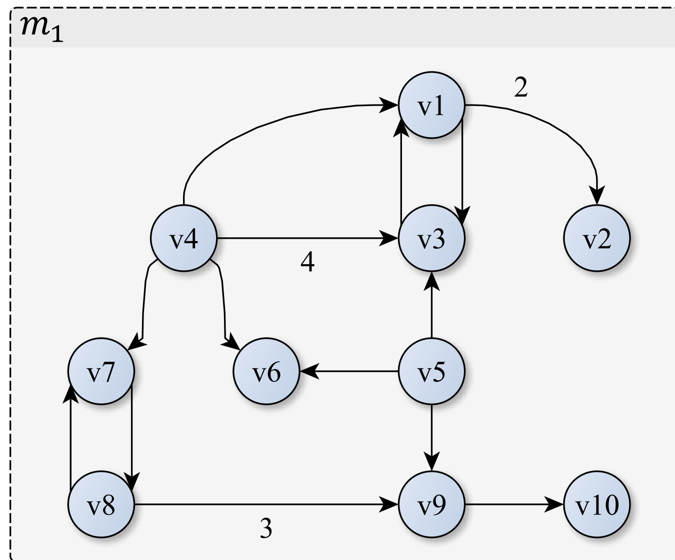


Figure 2.4 A trivial solution for the MDG of the software project represented in Figure 2.2 which contains only one module ($n = 1$).

2.1 Modularization Quality

Originally proposed in 1998, MQ is a family of quality metrics to evaluate the quality of modular organizations for the SMCP. The modularity value is calculated as a trade-off between coupling (to be minimized) and cohesion (to be maximized). In particular, two different implementations were proposed to compute the MQ of a modularization: BasicMQ and TurboMQ [95]. In both implementations, the higher the value, the better the solution.

2.1.1 BasicMQ

To describe the computation of BasicMQ, we must first define intra- and inter-connectivity. Intra-connectivity refers to the connectivity between components within the same module. Its objective is to measure cohesion. Therefore, the higher the intra-connectivity of the modularization, the better the solution. Specifically, the intra-connectivity of module m_i is defined as:

$$A_i = \frac{\mu_i}{|m_i|^2}, \tag{2.1}$$

where μ_i is the number of edges that connect vertices within the module m_i, and $|m_i|$ is the number of vertices belonging to module m_i. The set of edges connecting vertices within the same module is also known as the set of intracluster edges, formally defined as:

$$Intra(m_i) = \{\{u, v\} \in E : u, v \in m_i\}. \tag{2.2}$$

The intra-connectivity measure is a fraction of the maximum number of intracluster edges that module m_i can have ($|m_i|^2$). Therefore, the value of A_i is bounded in the range [0,1].

Inter-connectivity, as opposed to intra-connectivity, measures the connectivity between two different modules. Its aim is to measure coupling. Therefore, the lower the inter-connectivity of the modularization, the better the solution. Formally, the inter-connectivity between two modules m_i and m_j is defined as:

$$E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \frac{\varepsilon_{i,j}}{2 \cdot |m_i| \cdot |m_j|} & \text{if } i \neq j, \end{cases} \tag{2.3}$$

where $\varepsilon_{i,j}$ is the number of edges connecting vertices within m_i with vertices belonging to m_j, being the set of intercluster edges between modules m_i and m_j formally defined as:

$$Inter(m_i, m_j) = \{(u, v) \in E : u \in m_i \wedge v \in m_j \bigcup (u, v) \in E : u \in m_j \wedge v \in m_i\}. \tag{2.4}$$

Again, the inter-connectivity measure is a fraction of the maximum number of intercluster edges that could exist between modules m_i and m_j. Therefore, the value of $E_{i,j}$ is bounded in the range [0,1].

Once intra- and inter-connectivity measures have been defined, MQ is defined as a trade-off between the intra-connectivity of every module and the inter-connectivity between each pair of modules in the solution. Formally:

$$MQ = \begin{cases} \frac{1}{n} \cdot \sum_{i=1}^{n} A_i - \frac{1}{\frac{n(n-1)}{2}} \cdot \sum_{i,j=1}^{n} E_{i,j} & \text{if } n > 1 \\ A_1 & \text{if } n = 1. \end{cases} \tag{2.5}$$

As can be observed from Equation 2.5, MQ establishes a tradeoff between the average intra-connectivity and the average inter-connectivity. Therefore, the value of MQ is bounded in the range [-1,1]. Given an MDG with $|V|$ vertices and $|E|$ edges, the complexity of calculating BasicMQ is $O(|V|^2 \cdot |E|)$ [95].

2.1.2 TurboMQ

The TurboMQ measure was designed to improve BasicMQ in two ways: (i) supporting edge weights in the MDG and (ii) reducing the computational complexity to evaluate the quality of an MDG. Instead of calculating a trade-off between the average intra-connectivity and the average inter-connectivity of the whole solution, TurboMQ calculates the sum of the quality of each module.

First, a cluster factor CF is calculated for each module m_i as:

$$CF_i = \begin{cases} 0 & \text{if } \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} (\varepsilon_{i,j})} & \text{otherwise.} \end{cases} \tag{2.6}$$

It is important to notice that here, in contrast with BasicMQ, $\varepsilon_{i,j}$ does not represent the number of inter-cluster edges between modules m_i and m_j, but the sum of their weights. That is:

$$\varepsilon_{i,j} = \sum_{(u,v) \in Inter(m_i, m_j)} w(u,v). \tag{2.7}$$

Similarly, μ_i does not represent the number of intra-cluster edges within module m_i, but the sum of their weights. That is:

$$\mu_i = \sum_{(u,v) \in Intra(m_i)} w(u,v). \tag{2.8}$$

Then, TurboMQ is defined as:

$$\text{TurboMQ} = \sum_{i=1}^{n} CF_i. \tag{2.9}$$

The complexity of calculating TurboMQ is $O(|E|)$ [95], which is much lower than the complexity of calculating BasicMQ.

In Figure 2.5, we present the evaluation of the solution depicted in Figure 2.2. Next to each module, we detail the intra-connectivity (μ), the inter-connectivity (ε), and the CF of the module. For module m_1, $\mu_1 = 4$, since there are four edges with weight one that connect the vertices within m_1 ($(v4, v6), (v4, v7), (v7, v8)$, and $(v8, v7)$). Similarly, $\varepsilon_{1,2} = 5$, since there is one intercluster edge with weight equal to one ($(v4, v1)$) and another intercluster edge with weight equal to four ($(v4, v3)$). Therefore, $CF_1 = \frac{2 \cdot 4}{2 \cdot 4+5+4} = \frac{8}{17} = 0.47$. The rest of the CF values for each module are calculated in a similar fashion. Finally, the quality of the solution is calculated as TurboMQ $= CF_1 + CF_2 + CF_3 = 1.48$.

2.2 Function of Complexity Balance

The MQ problem has been widely studied by the SBSE community, but some researchers have highlighted some limitations of that approach. The Function of Complexity Balance (FCB) was proposed as an alternative to MQ with the aim of reducing the number of isolated modules (modules with only one component) [105]. Formally, FCB is described as follows:

$$\text{FCB} = \frac{C + \max\limits_{m_i \in M} (\mu_i)}{T}, \tag{2.10}$$

where C represents the coupling of the entire architecture and μ_i represents the cohesion of module m_i. In particular, C is calculated as follows:

$$C = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{\substack{(u,v) \\ \in \, Inter(m_i, m_j)}} w_{u,v}. \tag{2.11}$$

On the other hand, μ_i is calculated as described in Equation 2.8. Finally, T is the sum

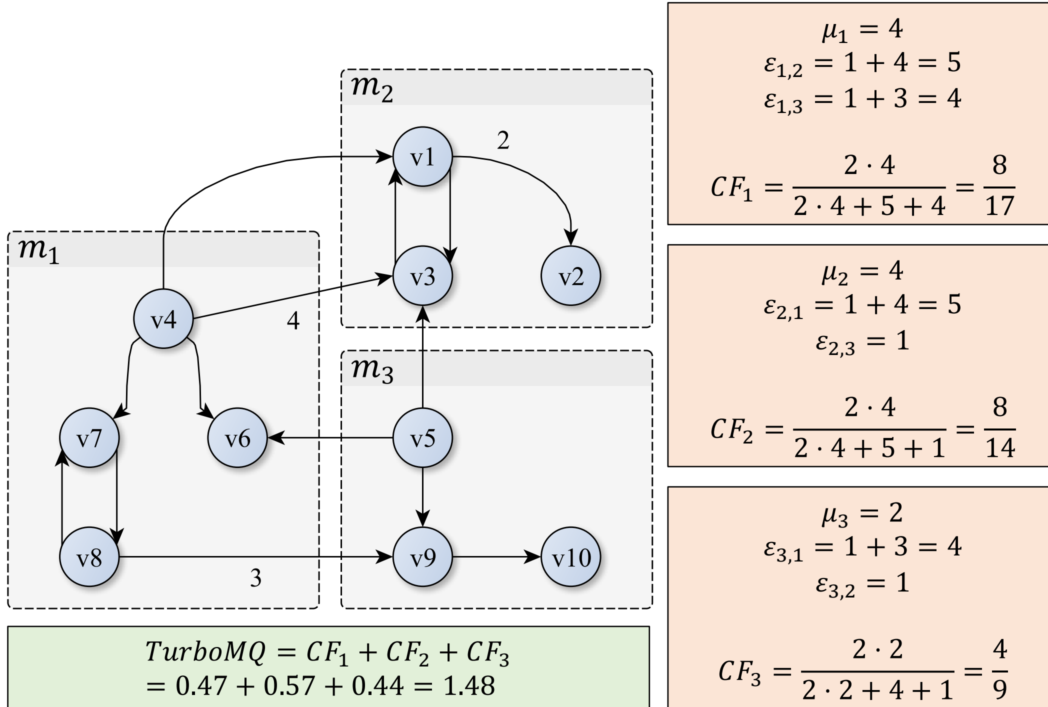


Figure 2.5 Calculation of the quality of a solution in the MQ problem for a modularized MDG of the software project represented in Figure 2.2 with three modules: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$.

of the weights of all edges of the entire architecture. Thus, T is a constant value that is independent of the particular clustering of the graph. Mathematically,

$$T = \sum_{e=(u,v) \in E} w_{u,v}. \tag{2.12}$$

This constant is used to normalize the resulting value of FCB in the range [0,1], allowing for comparisons between solutions for different software projects. In the case of the FCB problem, contrary to MQ, the lesser the value, the better the solution.

In Figure 2.6, we present the evaluation of the solution depicted in Figure 2.2. Next to each module, we detail its cohesion. For module m_1, $\mu_1 = 4$, since there are four edges with weight one that connect vertices within m_1 $((v4, v6), (v4, v7), (v7, v8)$, and $(v8, v7))$.

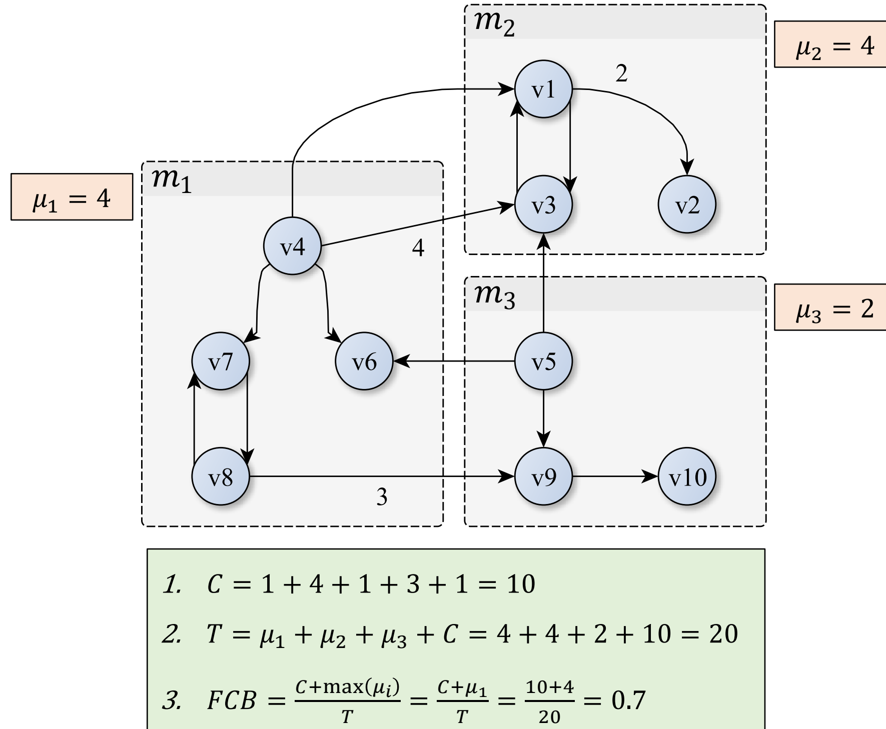


Figure 2.6 Calculation of the quality of a solution in the FCB problem for a modularized MDG of the software project represented in Figure 2.2 with three modules: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$.

Similarly, $\mu_2 = 4$, since there is one intracluster edge with weight two ($(v1, v2)$) and two intracluster edges with weight one ($(v1, v3)$ and $(v3, v1)$). Then, the coupling of the entire architecture (C) is calculated as the sum of weight of intercluster edges, which is equal to ten, since there are three intercluster edges with weight one ($(v4, v1), (v5, v3)$, and $(v5, v6)$), one edge with weight four ($(v4, v3)$), and one edge with weight three ($(v8, v9)$). Accordingly, T is equal to the sum of all weights, which is 20. Finally, the quality of the solution is calculated as $FCB = \frac{C+max(\mu_i)}{T} = \frac{14}{20} = 0.7$.

2.3 Maximizing Cluster Approach

When the principle of tight cohesion and loose coupling is taken to extremes, the best possible solution is trivial: a single module that contains all components. However, such a solution is not good for maintainability purposes. In practice, there exist other objectives that must be considered for the optimization of modularity in the context of SE. In the MQ problem, since quality is measured as the sum of the quality of each module, the number of modules is implicitly considered. In fact, the greater the number of modules, the easier it is to understand each module separately. However, the other trivial solution, where every component belongs to an isolated module, is not desirable either. In fact, one of the objectives of the FCB problem is to reduce the number of isolated modules that resulted in the case of MQ. Thus, a trade-off must be considered between the number of modules, the number of isolated modules, the cohesion, and the coupling of the solution.

In 2011, some authors considered the aforementioned problems and proposed a multi-objective approach for the SMCP [121]. By addressing the SMCP as an MOP, two advantages were obtained: (i) several conflicting objectives could be considered to better reflect the desires of software developers, and (ii) a set of non-dominated solutions could be provided to a decision maker (e.g., a software developer) to allow the introduction of their subjective experience and preferences into the process.

In particular, the authors proposed two different approaches. The first MOP was called Maximizing Cluster Approach (MCA). This approach considers five different objectives:

1. **Coupling**. The first objective is to minimize the coupling of the entire architecture. That is, to minimize the sum of the weights of the edges that connect vertices belonging to different modules (see Equation 2.11).

2. **Cohesion**. The second objective is to maximize the cohesion of the entire architecture. That is, to maximize the sum of the weights of the edges that connect vertices belonging to the same module. Formally:

$$\text{Cohesion} = \sum_{i=1}^{n} \mu_i, \tag{2.13}$$

   where μ_i is calculated as described in Equation 2.8.

3. **TurboMQ**. Interestingly, the MCA approach also includes MQ as one of the objectives to consider. This objective, to be maximized, is calculated as described in Equation 2.9.

4. **Number of modules**. In addition to the previous objectives, MCA considers the maximization of the number of modules in the solution. Trivially, the value of this objective is n.

5. **Number of isolated modules**. Finally, MCA also considers the minimization of the number of isolated modules. That is, the number of modules that contain only one vertex.

In Figure 2.7, we represent the evaluation of the solution depicted in Figure 2.2. Next to each module, we detail its cohesion. For module m_1, $\mu_1 = 4$, since there are four edges with weight one that connect vertices within m_1 $((v4, v6), (v4, v7), (v7, v8)$, and $(v8, v7))$. Similarly, $\mu_2 = 4$, since there is one intracluster edge with weight two $((v1, v2))$ and two intracluster edges with weight one $((v1, v3)$ and $(v3, v1))$. Then, the coupling of the entire architecture is calculated as the sum of the weights of intercluster edges, which is equal to ten, since there are three intercluster edges with weight one $((v4, v1), (v5, v3)$, and $(v5, v6))$, one edge with weight four $((v4, v3))$, and one edge with weight three $((v8, v9))$. Accordingly, the cohesion of the entire architecture is equal to ten, since $\mu_1 = 4$, $\mu_2 = 4$, and $\mu_3 = 2$. The TurboMQ value is equal to 1.48, as previously described in Figure 2.5, where the TurboMQ value of this solution is calculated. The number of modules (3) is calculated trivially. Finally, since there are no modules with only one vertex, the number of isolated modules is equal to zero.

2.4 Equal-size Cluster Approach

The Equal-size Cluster Approach (ECA) was the second problem introduced in [121], where MCA was also presented. ECA considers five different objectives. The first four objectives are shared with MCA: coupling, cohesion, TurboMQ, and the number of modules. However, instead of considering the number of isolated modules as the fifth objective,

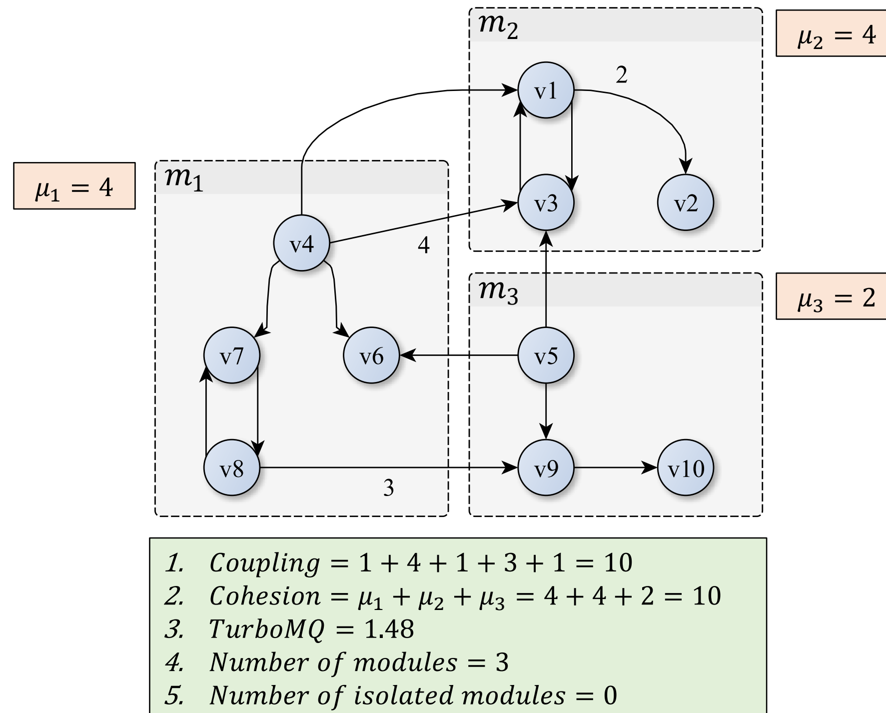


Figure 2.7 Calculation of the quality of a solution in the MCA problem for a modularized MDG of the software project represented in Figure 2.2 with three modules: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$.

ECA considers the minimization of the difference in size between the largest and smallest modules in the solution. Here, the size of a module is equal to the number of vertices belonging to that module. Therefore, the fifth objective is to minimize the difference between the maximum number of vertices in any module and the minimum number of vertices in any module. Due to the similarities between MCA and ECA, these two problems are often studied together.

In Figure 2.8, we present the evaluation of the solution depicted in Figure 2.2. Next to each module, we detail its cohesion. For module m_1, $\mu_1 = 4$, since there are four edges with weight one that connect vertices within m_1 $((v4, v6), (v4, v7), (v7, v8)$, and $(v8, v7))$. Similarly, $\mu_2 = 4$, since there is one intracluster edge with weight two $((v1, v2))$ and two

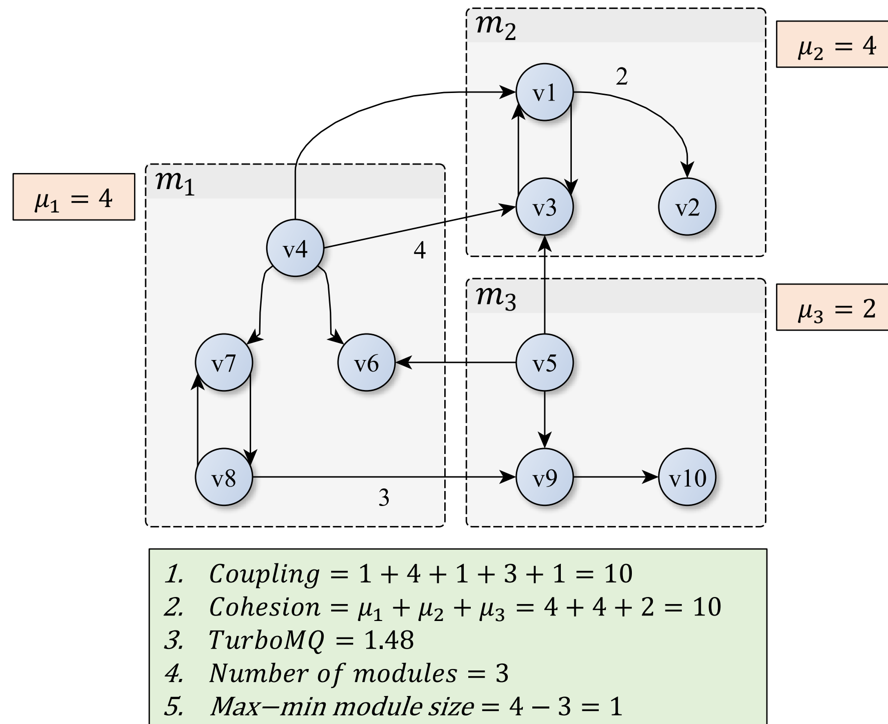


Figure 2.8 Calculation of the quality of a solution in the ECA problem for a modularized MDG of the software project represented in Figure 2.2 with three modules: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$.

intracluster edges with weight one ($(v1, v3)$ and $(v3, v1)$). Then, the coupling of the entire architecture is calculated as the sum of the weights of the intercluster edges, which is equal to ten, since there are three intercluster edges with weight one ($(v4, v1), (v5, v3)$, and $(v5, v6)$), one edge with weight four ($(v4, v3)$), and one edge with weight three ($(v8, v9)$). Accordingly, the cohesion of the entire architecture is equal to ten, since $\mu_1 = 4$, $\mu_2 = 4$, and $\mu_3 = 2$. The TurboMQ value is equal to 1.48, as previously described in Figure 2.5, where the TurboMQ value of this solution is calculated. The number of modules (3) is calculated trivially. Finally, the difference in size between the maximum number of vertices in any module and the minimum number of vertices in any module is equal to 1, since m_1 contains 4 vertices and m_3 contains 3 vertices.

Chapter 3

Literature review

Search-Based Software Engineering (SBSE) is a research area that focuses on both the reformulation of Software Engineering (SE) tasks as optimization problems and the design of optimization algorithms to solve them. Although there exist some preliminary works in the twentieth century that can be classified within the SBSE field [69, 70, 76, 112, 136, 138], the term Search-Based Software Engineering was first coined in 2001 by Mark Harman and Bryan F. Jones [54]. There, the authors claimed that SBSE was an emerging field of software engineering research and that they expected to "*see a dramatic development of the field*" [54]. More than twenty years later, in 2023, the Symposium on Search Based Software Engineering (SSBSE) celebrated its fifteenth edition [10]. Throughout the first eleven editions, more than 290 authors from 25 countries had already contributed to the main track of SSBSE [25]. The contributions of the SBSE community to reputed journals have also been significant. Only in Spain, more than 145 authors from more than 19 different institutions have published their results on a variety of SBSE topics [125].

In the SBSE field, there exist different families of optimization problems, each targeting a different SE task. Following the SDLC structured approach, SBSE problems can be categorized into different groups, depending on the software development phase in which these problems arise. Following this idea, in Table 3.1 we present a classification of different SBSE problems in the phase of a SDLC where they arise primarily, based on the related literature [35, 55, 102, 124, 129, 131]. As can be observed, SBSE problems arise in many SDLC phases, from project management and requirements to software maintenance.

Table 3.1 Classification of some families of problems identified in the SBSE research field according to the phase within a SDLC where they mostly arise.

SDLC phase	SBSE problem	Relevant references
Project management and requirements	Predictive modeling Project scheduling Requirements selection and prioritization Next release problem	[45, 117, 130, 36]
Software analysis and design	Object-oriented architecture design Software product lines Service-oriented architecture design Model-based software engineering	[83, 26, 127]
Integration and deployment	Service composition Component allocation Software deployment Systems configuration	[126, 11]
Software testing	Test case generation Test case prioritization Black-box testing Regression testing Program verification	[74, 93, 53, 47]
Software maintenance	Software modularization Automated bug fixing Software refactoring	[102, 17, 60, 141]

The family of problems that focus on software modularization, located in the maintenance phase, is also commonly known as the SMCP. The SMCP is a family of optimization problems whose objective is to find the best possible organization for a software project in terms of modularity. As defined by the ISO [63], modularity is the set of "*software attributes that provide a structure of highly independent components*". In a desirable modular structure, components within the same module are strongly related among them (high cohesion) and weakly connected to the components of other modules (low coupling). The main goal of the SMCP is to facilitate the understanding of software projects.

In Table 3.2 and Table 3.3, we present a chronological summary of articles that study mono-objective and multi-objective SMCP problems. The first table presents works that

tackle single-objective variants, while the second one presents works that study multi- and many-objective variants. For each work included in the table, we report the year of publication, the problem studied, and the metaheuristic framework on which the approach is based. Population-based heuristics are highlighted in green, while trajectory-based heuristics are highlighted in red. As can be observed in Table 3.2, the first approach towards the SMCP was proposed by Mancoridis et al. in 1998 [89]. These authors proposed the MQ objective function, where the value of modularity is calculated as a trade-off between coupling and cohesion. This metric was later extended into two variants to accelerate its computation and improve the results: BasicMQ [94] and TurboMQ [95]. As can be seen in the table, MQ has become the most studied problem in the SMCP literature. However, some recent studies have highlighted some concerns about its design [16, 66]. Due to these reasons, alternative variants have been proposed in the literature, including: Dependency Quality and Connection Quality (DQCQ) [3]; Entropy-based Objective Function (EOF) [66]; FCB [105]; Modularization Quality measure based on similarity (MS) [56]; Cohesion, Coupling, package Count index, and package Size index (CCCS) [7]; Linear Compound Criteria (LCC) [119]; and MQ, Non-extreme distribution, Coupling, and Cohesion (MNCC) [123].

Traditionally, software engineers have organized their projects in modules based on their own expertise and understanding of the code. Consequently, this process is often neither systematic nor repeatable [16]. Due to this concern, some authors have proposed multi-objective approaches for the SMCP, which seem to be more suitable in this context because: i) the consideration of different conflicting objectives is a more accurate reflection of the modularity of the system than the consideration of each objective in isolation; and ii) presenting a set of good solutions to a decision maker (e.g., a software developer) allows the stakeholders to prioritize some objectives over others depending on the context, and also to introduce their subjective experience in the process.

Following the aforementioned ideas, Praditwong et al. [121] introduced two different multi-objective problems for the SMCP in 2011: MCA and ECA. These variants became the most studied multi-objective variants of the SMCP in the literature, and were also extended into two additional variants: the Extended Maximizing Cluster Approach (E-MCA) and the Extended Equal-size Cluster Approach (E-ECA) [24]. As can be observed in Table 3.3, other multi-objective approaches to the SMCP have also been studied throughout the

years, such as: Structure of packages, Semantics coherence, and History of changes (SSH) [99]; Multi-Factor Module Clustering (MFMC) [59]; Interactive Fitness Function (IFF) [128]; and Multi-Objective Fitness function (MOF) [67]. However, none were as popular as MCA and ECA.

Regardless of the problem studied, many different metaheuristics have been proposed for the SMCP. It is worth mentioning that, since the SMCP is proven to be NP-complete [18], exact methods are not suitable for the problem, except for tiny software projects [95]. Instead, approximate search-based metaheuristics are more convenient [25, 55, 129]. In this sense, population-based methods have traditionally been favored in the literature for the SMCP, including: Genetic Algorithms (GAs) [37, 68, 85, 87, 121]; Hybrid Genetic Algorithms (HGAs) [87, 105]; Genetic Algorithm with Hill Climbing (GAHC) [85]; Firefly Algorithm (FA) [86]; Evolutionary Call-Dependency Graph Modularization method (E-CDGM) [65]; Estimation of Distribution (EoD) [67, 133]; Harmony Search (HS) [7]; Multi-Agent Evolutionary Algorithms (MAEAs) [56, 57]; Multi-objective Hyper-heuristic Evolutionary Algorithm (MHypEA) [78]; Two-Archive Artificial Bee Colony (TA-ABC) [8]; Many-objective Artificial Bee Colony (MaABC) [24]; Interactive Evolutionary Computation (IEC) [128]; Particle Swarm Optimization (PSO) [123]; Grid-based Large-scale Many-objective Particle Swarm Optimization (GLMPSO) [122]; and Non-dominated Sorting Genetic Algorithm III (NSGA-III) [99]. In contrast, some authors have remarked the absence of efficient single-solution metaheuristics such as Iterative Local Search (ILS) or Variable Neighborhood Search (VNS) from the literature [129]. To fill this gap, multiple trajectory-based metaheuristics have been proposed as an effective alternative to the classic population-based proposals, achieving competitive results. Most of the first proposals in this sense were based on Hill Climbing (HC) [73, 85, 88, 89, 94, 96] or Simulated Annealing (SA) [3, 97, 98]. More recently, some algorithms have been proposed based on graph properties, such as the Graph-based Modularization Algorithm (GMA) [119], or based on metaheuristics such as Large Neighborhood Search (LNS) [104], VNS [144, 145], Multi-Objective Variable Neighborhood Descent (MO-VND) [143], or Greedy Randomized Adaptive Search Procedure (GRASP) and Variable Neighborhood Descent (VND) [141].

Table 3.2 Chronological summary of proposals for single-objective optimization in the SMCP family.

Year	Ref.	Single-objective optimization problems							
		MQ	DQCQ	MS	CCCS	MNCC	EOF	FCB	LCC
1998	[89]	HC	-	-	-	-	-	-	-
1999	[88]	HC	-	-	-	-	-	-	-
	[37]	GA	-	-	-	-	-	-	-
2001	[94]	NAHC, SAHC	-	-	-	-	-	-	-
2002	[96]	SAHC	-	-	-	-	-	-	-
2005	[85]	HC, GAHC	-	-	-	-	-	-	-
2006	[97]	GA, HC	-	-	-	-	-	-	-
2008	[98]	SA	-	-	-	-	-	-	-
2009	[3]	-	SA	-	-	-	-	-	-
	[87]	HGA	-	-	-	-	-	-	-
2011	[120]	GA	-	-	-	-	-	-	-
2014	[116]	ILS	-	-	-	-	-	-	-
	[86]	FA	-	-	-	-	-	-	-
2016	[56]	-	-	HC, GA, MAEA	-	-	-	-	-
	[65]	E-CDGM	-	-	-	-	-	-	-
	[68]	GA	-	-	-	-	-	-	-
	[133]	EoD	-	-	-	-	-	-	-
2017	[57]	MAEA	-	-	-	-	-	-	-
	[7]	-	-	-	HS	-	-	-	-
	[73]	HC	-	-	-	-	-	-	-

Table 3.2 Chronological summary of proposals for single-objective optimization in the SMCP family.

Year	Ref.	Single-objective optimization problems							
		MQ	DQCQ	MS	CCCS	MNCC	EOF	FCB	LCC
2018	[123]	-	-	-	-	PSO	-	-	-
	[104]	LNS	-	-	-	-	-	-	-
2019	[66]	-	-	-	-	-	GA	-	-
2020	[105]	-	-	-	-	-	-	HGA	-
2021	[119]	-	-	-	-	-	-	-	GMA
2022	[141]	GRASP-VND	-	-	-	-	-	-	-
	[144]	-	-	-	-	-	-	VND	-
2024	[145]	-	-	-	-	-	-	GVNS	-

Table 3.3 Chronological summary of proposals for multi-objective optimization in the SMCP family.

		Multi-objective optimization problems					
Year	Ref.	MCA, ECA	SSH	MFMC	IFF	E-MCA, E-ECA	MOF
2011	[121]	GA	-	-	-	-	-
2015	[99]	-	NSGA-III	-	-	-	-
2016	[78]	MHypEA	-	-	-	-	-
2017	[59]	-	-	HC	-	-	-
2018	[128]	-	-	-	IEC	-	-
	[8]	TA-ABC	-	-	-	-	-
	[24]	-	-	-	-	MaABC	-
2019	[67]	-	-	-	-	-	EoD
2022	[9]	GA	-	-	-	-	-
	[122]	-	-	-	-	GLMPSO	-
	[143]	MO-VND	-	-	-	-	-

As can be observed in Table 3.2, there exist many algorithmic proposals for the problems belonging to the SMCP family. Since in this doctoral thesis we study four problems in the SMCP family, in the following sections we review the best methods identified in the literature for these problems. In particular, we review a LNS method proposed for the MQ problem (Section 3.1), a HGA method proposed for the FCB problem (Section 3.2), and a TA-ABC method proposed for the MCA and ECA problems (Section 3.3).

3.1 Large Neighborhood Search for the MQ problem

In 2018, some authors proposed a method based on Large Neighborhood Search (LNS) to tackle the TurboMQ problem [104]. Intuitively, the higher the number of solutions explored, the greater the chance to find the global optimum for the problem and the instance at hand. Therefore, a search in very large neighborhoods should lead to a better solution than exploring small neighborhoods. However, it is often the case that a trade-off must be accepted between the quality of the solution and the computing time. LNS belongs to a family of heuristics known as Very Large Scale Neighborhood search (VLSN) algorithms. The idea of VLSN algorithms is to explore a large neighborhood only considering a restricted subset of the solutions in the neighborhood. LNS was proposed by Shaw in 1998 [132]. In LNS, an initial solution is gradually improved by iteratively destroying and repairing the solution.

In Algorithm 1, the pseudocode of LNS is presented. The algorithm receives an initial solution x. This solution is saved as the best solution x_b (step 2). Then, x is destroyed and repaired (steps 4-5). The resulting solution, x'', is then evaluated twice. First, if the acceptance criteria are met, it is saved to continue improving it in the next iteration (steps 6-7). This acceptance criteria can be established to allow for temporary deterioration of the current solution in order to escape from local optima. Finally, the quality of the resulting solution is calculated. If the current solution x'' is better than the best solution found x_b, then it is saved as the best solution (steps 8- 9). This process is repeated iteratively until the stopping criterion is met (step 3). Once the stopping criterion has been met, the procedure ends and the best solution found (x_b) is returned.

As can be observed, the pseudocode of LNS contains some components that must be

Algorithm 1: LNS procedure

```
1  Procedure LNS(x):
2      x_b ← x
3      do
4          x' ← destroy(x)
5          x'' ← repair(x')
6          if accept(x'', x) then
7              x ← x''
8          if OF(x'') > OF(x_b) then
9              x_b ← x''
10     while end condition is not met
11 return x
```

designed for the particular problem at hand (e.g., destroy, repair, accept, stopping criterion, etc.). The authors proposed several heuristics for the configuration of the components and performed a set of preliminary experiments to choose the best configuration. Here, we describe the final configuration selected by the authors.

First, to construct an initial solution for the LNS algorithm, the authors proposed an agglomerative method. This method starts by placing each vertex in a separate module. Then, at each iteration, all possible merges between modules are evaluated and the best (in terms of MQ) is performed. The method continues to perform merges at each iteration until all vertices are placed in the same module. Then, the best solution found during the process is returned.

For the destructive component of the algorithm, the authors proposed a random approach. This approach removes k vertices from the solution, selected at random. Here, k is a parameter that can be adjusted. In particular, k represents the degree of destruction. In this case, the degree of destruction was set to $k = 0.1 \cdot n$. That is, the `destroy` method removes 10% of the vertices in the solution at hand.

For the repair component of the algorithm, two approaches were proposed. First, a Repair Greedy Best Improvement Random was described. This method performs an iteration for each vertex removed by the `destroy` method. At each iteration, one of the removed vertices is selected at random. Then, its insertion into every module in the solution is evaluated, also considering the creation of a new isolated module, and the insertion that leads

to the highest value of MQ is performed. The second repair method is regarded as Repair Greedy Best Improvement. In this case, the insertion of all the removed vertices into the solution is considered simultaneously and the best assignment in terms of MQ is performed.

For the stopping criterion, the authors considered a maximum number of iterations without improvement. In particular, the maximum number of iterations without improvement was set to 2000. Moreover, at each iteration, only one of the aforementioned repair methods is used. The repair method to be used is changed when 1000 iterations have been performed without improving the best solution.

Finally, for the acceptance criterion, the proposed heuristic only accepts solutions that improve the value of the objective function (i.e., no deterioration is allowed).

The authors favorably compared the proposed approach with a ILS procedure, considered the best for the MQ [104, 116] problem. Therefore, the LNS procedure presented by the authors became the most competitive algorithm for MQ. In an effort to allow for the reproducibility of the experiments, the authors reported the detailed results per instance and published the source code online.

3.2 Hybrid Genetic Algorithm for the FCB problem

The Hybrid Genetic Algorithm (HGA) described in this section was proposed in 2020 for the FCB problem [105]. The proposed method consists of the hybridization of a constructive heuristic to generate initial high-quality solutions and a GA.

The proposed constructive heuristic is similar to the one described for LNS in the previous Section 3.1. First, every vertex is placed in an isolated module. Then, an iterative process is performed, where two modules are merged at each iteration. In particular, all possible merges of existing modules are evaluated at each iteration and the best merge, in terms of the resulting FCB value, is performed. The process ends when all vertices have been placed in the same module. At the end, the best solution found during the process is returned. In the case of ties between different merge options, one of the operations is selected at random. This makes it possible for the constructive heuristic to return different solutions after each execution.

The constructive heuristic is used to generate an initial population of solutions that is

then improved with a GA. The proposed GA uses two well-known operators: the Laplace crossover [32] and the Power mutation [33]. Both operators are available in Matlab, where the authors implemented the proposed approach. For the GA parameters, the population size is set to $max(min(10 \cdot |V|, 200), 40)$ and an elitist archive is set to have a size equal to $0.05 \cdot max(min(10 \cdot |V|, 200), 40)$. That is, the population size is set in the range [40,200], depending on the number of components, and the size of the elitist archive is set to be a 5% of the population size. In addition, two hybrid stopping criteria are set: a maximum of 400 generations and a maximum of 200 generations without improvement. If either is reached, the GA stops. The selection function and the fraction of the population that is used for crossover and mutation are not reported in the article.

In order to improve the efficiency of the designed algorithm, the authors proposed two advanced strategies. The first strategy consists of a vectorization method to speed up the evaluation of possible merges in the agglomerative constructive. This method is based on a matrix representation to evaluate possible merges. Although the method is more complex than the intuitive approach (to evaluate the merges in a loop), it avoids repeated evaluation and accelerates the constructive procedure.

The second strategy devised by the authors also consists of a vectorization method, but this time designed to evaluate the fitness function of a given solution. In particular, the authors state that the most critical part of evaluating the FCB value is calculating the cohesion of each module. Therefore, they propose a vectorization method based on a matrix representation to calculate the cohesion value of every module in the solution.

Finally, the authors compare the performance of the proposed HGA approach with two reference algorithms: a GA [137] and a multi-start HC algorithm [98]. The comparison is made over a set of sixteen real software systems and a set of synthetic instances generated by the authors [105]. Unfortunately, neither the code nor the datasets used are publicly available.

3.3 Two-Archive Artificial Bee Colony for the MCA and ECA problems

In 2018, Chhabra et al. published a Two-Archive Artificial Bee Colony (TA-ABC) algorithm for the multi-objective MCA and ECA problems, previously described in Section 2.3 and Section 2.4, respectively. Swarm intelligence is broadly defined as the collective behavior of self-organized and decentralized swarms [72]. The Artificial Bee Colony (ABC) algorithm is a swarm-based metaheuristic that was introduced by Karaboga in 2005 [71], inspired by the behavior of bee hives.

The pseudocode of the proposed method is presented in Algorithm 2. The algorithm is divided into four phases: population initialization, employed bees, onlooker bees, and scout bees. The procedure receives three parameters: the size of the population (PS), the number of maximum iterations (NI), and the maximum number of attempts to improve any given solution (LMT). First, the population is randomly initialized (steps 4). To generate each solution, every vertex is placed in a module randomly chosen in the range [1,$|V|$]. Then, in the Employed Bees phase, each solution is modified by changing one decision variable (step 7). That is, by moving one vertex from its current module to another. If the resulting solution dominates the original one, then the modified solution replaces the original in the population FS. In the Onlooker Bees phase, the same procedure is repeated, but only considering some of the solutions, depending on their fitness (step 8). Here, the fitness of each solution is calculated on the basis of the indicator $I_{\varepsilon+}$ given in Indicator-Based Evolutionary Algorithm (IBEA) [148]. At each iteration of this phase, a solution is selected by the roulette wheel selection mechanism and then modified. Again, if the resulting solution dominates the original one, then the modified solution replaces the original in the population FS. Finally, in the Scout Bees phase, solutions that have not been improved after a certain number of attempts are replaced by new solutions generated at random (step 9). As can be noticed, employed and onlooker bees are responsible for the intensification of the search, whereas scout bees provide some diversification. At the end of each iteration, some of the solutions in FS are stored in two different archives: a Convergence Archive (CA) and a Diversity Archive (DA). These archives contain non-dominated solutions and implement a mechanism to maintain diverse solutions. Regarding the stopping criterion, the authors

define a maximum number of evaluations for comparison purposes.

Algorithm 2: Artificial Bee Colony procedure

```
1  Procedure ABC(PS, NI, LMT):
2      CA ← ∅
3      DA ← ∅
4      FS ← PopulationInitialization()
5      iter ← 0
6      while iter < NI do
7          FS ← EmployedBeesPhase(PS, FS)
8          FS ← OnlookerBeesPhase(PS, FS, NI)
9          FS ← ScoutBeesPhase(PS, FS, LMT)
10         CA, DA ← UpdateArchives(FS, CA, DA)
11         iter ← iter + 1
12     end
13 return CA ∪ DA
```

The proposed algorithm was tested on ten weighted and seven unweighted real software instances. The smallest instance had a minimum of 20 vertices and 57 edges, while the largest instance had 198 vertices and 3262 edges.

The performance of the algorithm was compared with two state-of-the-art methods: the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [30] and a Two-Archive Algorithm (TAA) [121]. The configuration of these methods is the same as that reported in [15, 121]. According to the results, TA-ABC performed better than NSGA-II and TAA considering several quality metrics, including Hypervolume, Inverted Generational Distance (IGD), and Spread.

Chapter 4

Algorithmic proposal

In this doctoral thesis, four different problems are studied: TurboMQ, FCB, MCA, and ECA. To address these problems, three different algorithms are proposed. For the first problem, MQ, a hybridization between the GRASP and the VND metaheuristics is proposed. This proposal is described in Section 4.2. For the second problem, FCB, a method based on the General Variable Neighborhood Search (GVNS) scheme is proposed. This method is described in Section 4.3. Finally, for the third and fourth problems, MCA and ECA, a method based on the Multi-Objective General Variable Neighborhood Search (MO-GVNS) methodology is proposed. This approach is described in Section 4.4. Before describing the methods mentioned above, we first introduce some of the ideas that underlie the foundations of GRASP, VNS, and Multi-Objective Variable Neighborhood Search (MO-VNS) in Section 4.1. All the methods proposed in this doctoral thesis include some components based on the exploration of neighborhood structures. The neighborhood structures proposed for the different problems studied are described in Section 4.5. Finally, some advanced strategies are proposed to improve the efficiency of the procedures proposed in this doctoral thesis. These strategies are described in Section 4.6.

4.1 Fundamentals of the algorithmic methodologies

The algorithms proposed in this doctoral thesis are based on the GRASP [43, 44] and VNS [100] methodologies. In this section, we provide an introduction to these schemes

and outline their main ideas and foundations. First, we introduce the GRASP scheme in Section 4.1.1. Then, we describe the main ideas and concepts of VNS in Section 4.1.2. Finally, we present the MO-VNS framework in Section 4.1.3. Although MO-VNS is an extension of VNS, given that it is designed to tackle MOPs, we decide to describe the MO-VNS scheme in a separate section.

4.1.1 Greedy Randomized Adaptive Search Procedure

The GRASP methodology was first presented by Thomas A. Feo and Mauricio G.C. Resende in 1989 [43, 44]. GRASP is a multistart procedure. At each iteration, two phases are performed: construction and local search. In the construction phase, a feasible initial solution is built. This solution is then improved in the local search phase until a local optimum is found. After a local optimum has been reached, the process is restarted and another solution is built and improved. The construction phase of GRASP starts from an empty solution and adds one element at each iteration until the solution is complete. At each step, the element to be added to the solution is selected following a semi-greedy criterion. The decision is taken at random among the best available elements according to the greedy criterion. That is why GRASP is greedy and randomized.

In Algorithm 3, we show the pseudocode of GRASP. This procedure receives one parameter: the maximum number of iterations (*MaxIter*). First, the best solution found during the search is initialized (step 2). Then, at each iteration (steps 4-11), an initial solution is built (step 5) and improved (step 8). Moreover, if the built solution is not feasible (step 6), it is necessary to repair it before performing the local search (step 7). At the end of each iteration, the local optimum obtained (x') is compared with the best solution found during the search (step 9). If the new solution x' is better than the previous best solution, then it is saved as the new best solution (step 10). Finally, the best solution found during the search is returned.

In Algorithm 4, we show the pseudocode of the constructive procedure in GRASP. The algorithm starts by initializing a solution x, which is empty (step 2), and the list of candidates CL, which will contain the set of elements that can be added to the solution (step 3). Then, the value of α is selected at random in the range [0,1] (step 4). Once these

Algorithm 3: Pseudocode of GRASP

```
1  Procedure GRASP(MaxIter):
2      bestSolution ← ∅
3      i ← 0
4      while i < MaxIter do
5          x ← Construction()
6          if x is not feasible then
7              x ← Repair(x)
8          x' ← LocalSearch(x)
9          if IsBetter(x',bestSolution) then
10             bestSolution ← x'
11     end
12 return bestSolution
```

variables have been initialized, the procedure iteratively adds an element to the solution until the solution is complete (steps 5-11). At each iteration, a threshold is first established (step 6). This threshold is in the range $[lb,ub]$, where lb is the benefit of choosing the worst possible element from the CL and ub is the benefit of choosing the best possible element from the CL, according to a greedy function g. Then a Restricted Candidate List (RCL) is built that contains only the candidates from the CL that are better than the aforementioned threshold (step 7). As can be seen, the higher the value of α, the higher the threshold and the more restrictive the list. Next, a candidate element s is selected at random from the *RCL* (step 8) and added to the current solution x (step 9). Furthermore, the chosen candidate s is removed from the *CL* (step 10). Finally, the procedure returns the built solution.

In Figure 4.1, we illustrate the main idea of the GRASP constructive procedure. In particular, we represent a *CL* and a *RCL* in one iteration of the constructive procedure. In the y-value, we represent candidate elements that can be added to a given solution. In the x-axis, we represent the value of each candidate u according to the greedy function g. As can be observed, a threshold *th* is established in the figure to delimit the set of candidates that are considered in the *RCL*. This threshold depends on the value of α, the value of the worst candidate according to the greedy function ($min(g(u))$), and the value of the best candidate according to the greedy function ($max(g(u))$). At the end of the iteration, a candidate from the *RCL* will be selected and added to the solution. The *RCL* is the greedy aspect

Algorithm 4: Pseudocode of the construction phase in GRASP

```
1  Procedure GRASPConstructive():
2      x ← ∅
3      CL ← GetCandidates(x)
4      α ← Random(0.0, 1.0)
5      while CL ≠ ∅ do
6          th ← min(g(u)) + α · (max(g(u)) − min(g(u))) | u ∈ CL
7          RCL ← {u ∈ CL | g(u) ≥ th}
8          s ← RandomChoice(RCL)
9          x ← x ∪ s
10         CL ← CL \ s
11     end
12 return x
```

of GRASP, since candidates are selected greedily. The selection of an element from the *RCL* is the probabilistic aspect of GRASP, that provides diversification to the constructive procedure. Once the selected element is incorporated into the partial solution, the candidate list is updated, and the costs are reevaluated. This is the adaptive aspect of GRASP [50].

In the pseudocode presented, the value of α is randomly obtained from a uniform distribution for each construction. However, other strategies can be implemented. For example, α can be fixed beforehand for all constructions in the procedure, it can be obtained randomly from non-uniform distributions, or it can be self-tuned along the search process in adaptive schemes.

The greedy function g used to evaluate each candidate in the CL to the solution is problem-dependent. In addition, for some problems, adding the selected element to the solution is not a trivial task. In those cases, it might be necessary to design a method to decide how to add the selected element to the solution at hand.

GRASP is one of the most well-known methodologies in heuristic research. It has been applied to a wide variety of applications and several extensions and hybrid schemes exist. For more information on the subject, we refer the reader to relevant sources [43, 44, 50, 91].

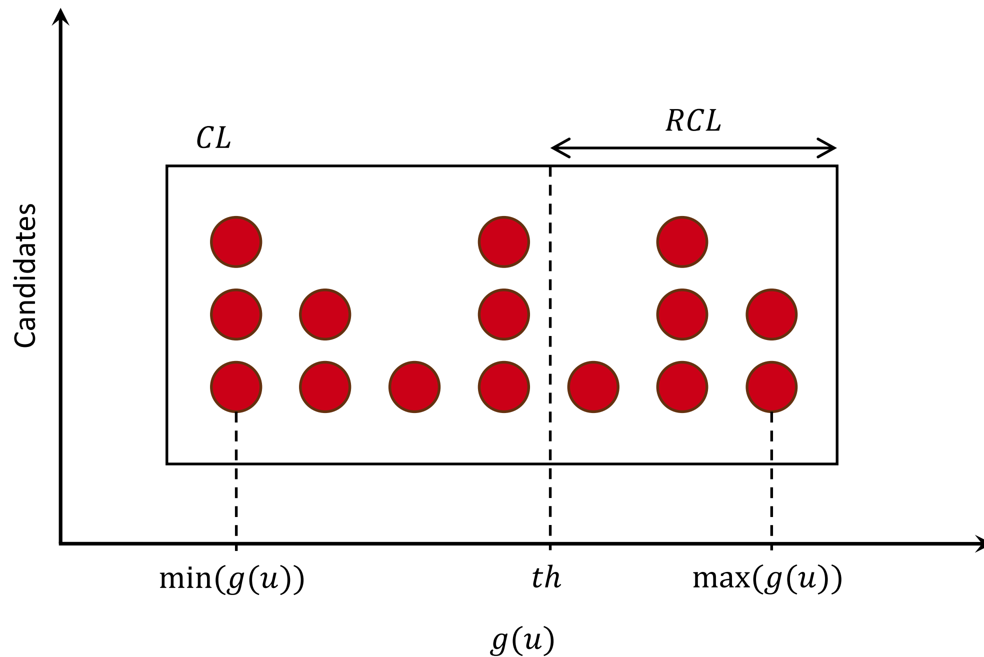


Figure 4.1 Restricted Candidate List in the GRASP constructive procedure (adapted from [38]).

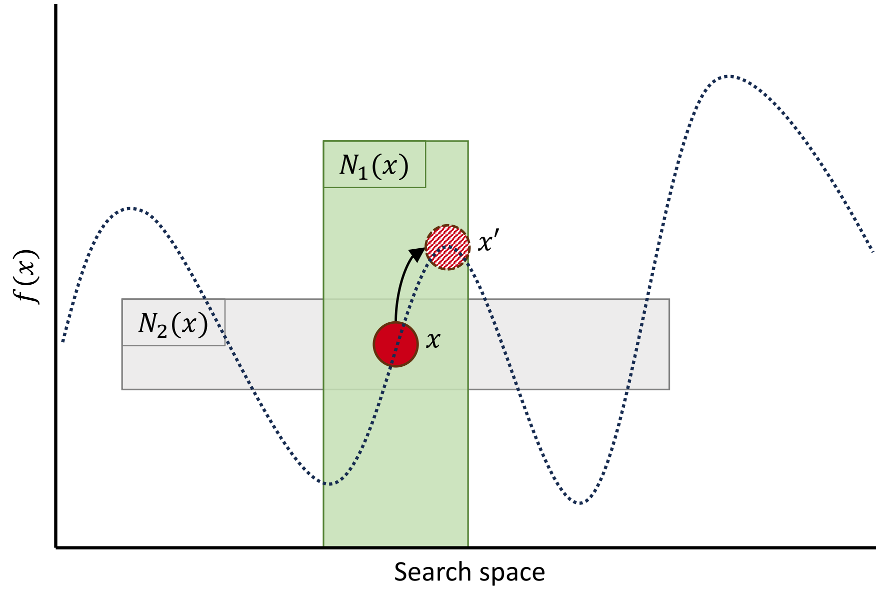
4.1.2 Variable Neighborhood Search

The VNS methodology was first proposed in 1997 by Nenad Mladenović and Pierre Hansen [100]. The main idea of VNS is to perform a systematic change of the neighborhood structure to explore during the search process. The benefits of exploring different neighborhood structures in VNS instead of a single one are based on three facts: (i) "*a local minimum within one neighborhood structure is not necessarily so for another*" [50]; (ii) "*a global minimum is a local minimum within all possible neighborhood structures*" [50]; and (iii) "*for many problems, local minima within one or several neighborhoods are relatively close to each other*" [50].

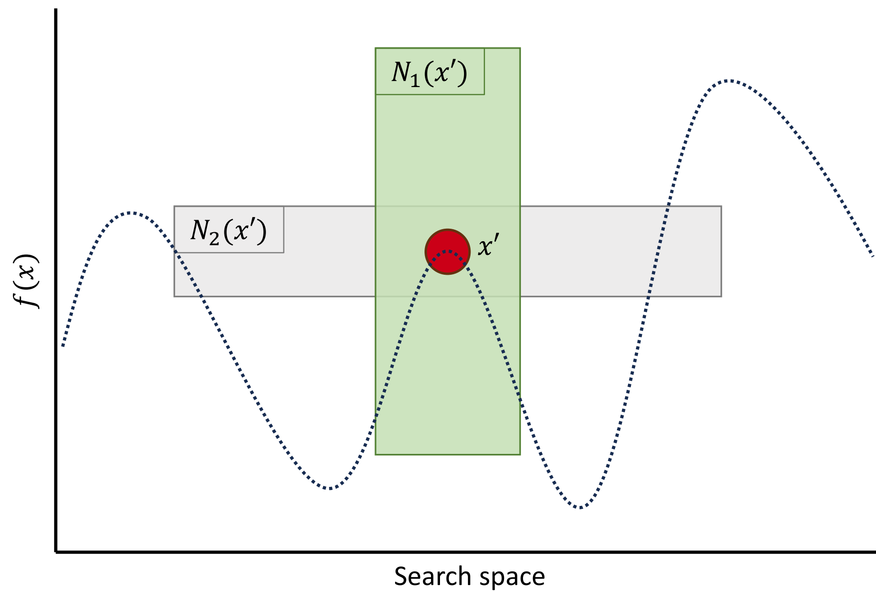
In Figure 4.2, we illustrate some of the main ideas behind the VNS framework. In particular, we illustrate some iterations of a search process based on the ideas of VNS. In Figures 4.2(a), 4.2(b), 4.2(c), and 4.2(d), we depict the set of possible solutions for a given optimization problem in a plane: the x-axis represents the search space, while the y-axis represents the objective function value of each possible solution. In Figure 4.2(a),

we represent an initial solution x with a solid red circle. Two rectangles represent the areas of the search space included in the neighborhood structures N_1 (depicted in green) and N_2 (depicted in gray) of the initial solution x. In this first illustration, $N_1(x)$ is explored, resulting in the application of a move operator to x in order to obtain solution x', the best solution in the neighborhood, which is represented by a red circle with a diagonal pattern. Next, in Figure 4.2(b), we represent the exploration of $N_1(x')$, where x' is the solution obtained in the previous figure. As can be observed, there is no neighbor solution that is better than x' in $N_1(x)$. Then, an alternative neighborhood structure, N_2, is explored in Figure 4.2(c). Although x' was a local optimum in $N_1(x')$, it is not a local optimum in the neighborhood structure $N_2(x')$. As a result, x'' is obtained. Finally, in Figure 4.2(d), N_1 is explored again. After the previous move operation, the current solution x'' is no longer a local optimum in N_1, and x''' is obtained. This solution, as can be observed, is a global optimum in the search space represented in Figure 4.2(d) and therefore a local optimum within all possible neighborhood structures. Therefore, the search process is finished.

There exist several schemes within the VNS methodology that mix stochastic and deterministic behaviors, such as Basic Variable Neighborhood Search (BVNS), Variable Neighborhood Descent (VND), Reduced Variable Neighborhood Search (RVNS), or General Variable Neighborhood Search (GVNS) [91]. In Algorithm 5, we present the pseudocode of a BVNS procedure, the first scheme proposed in the VNS framework [100]. This method includes both deterministic and stochastic components. As can be observed, the method receives three parameters: an initial solution (x), a maximum perturbation size (k_{max}), and a time limit (t_{max}). The procedure will explore the search space until the time limit t_{max} is reached (steps 2-14). First, the variable k is initialized (step 4) and an iterative process is started (step 5). At each iteration, a new solution x'' is generated by performing a shake procedure (step 6) and a local search (step 7) to solution x. The shake procedure performs a perturbation in the incumbent solution by performing random moves within a neighborhood structure. The number of moves (i.e., the size of the perturbation) to be performed depends on the value of the variable k. The objective of the shake procedure is to escape from local optima, introducing some diversification in the search process. The local search procedure explores a given neighborhood structure, performing moves that improve the incumbent solution. If the resulting solution is better than the best solution found (step 8),

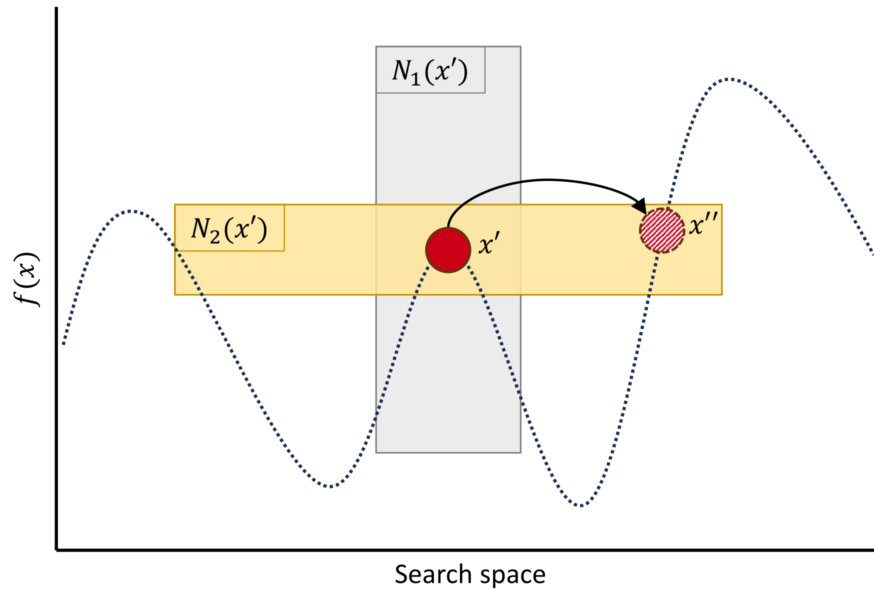


(a) Exploration of neighborhood structure $N_1(x)$, which results in obtaining solution x'.

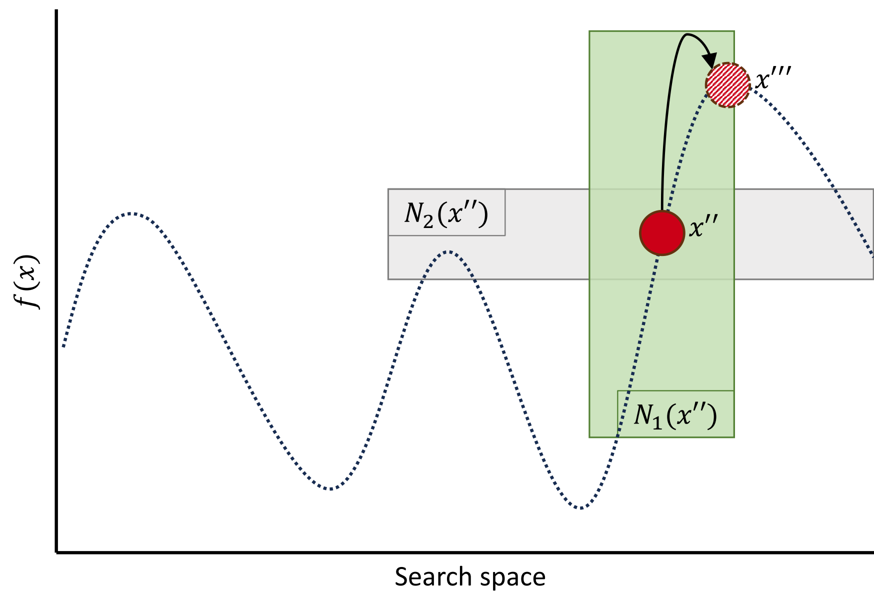


(b) Exploration of neighborhood structure $N_1(x')$, where x' is a local optimum.

Figure 4.2 Illustration of some of the main ideas behind the VNS framework.



(c) Exploration of neighborhood structure $N_2(x')$, which results in obtaining solution x''.



(d) Exploration of neighborhood structure $N_1(x'')$, which results in obtaining solution x'''.

Figure 4.2 Illustration of some of the main ideas behind the VNS framework.

then it is saved (step 9) and the search is restarted from the first neighborhood structure by setting k to one (step 12). Otherwise, k is incremented to try to escape from local optima in the next iteration (step 10). This process is repeated until the maximum size of the perturbation (k_{max}) is reached (step 5). Finally, the method returns the best solution found during the search.

Algorithm 5: Pseudocode of the BVNS scheme

```
1  Procedure BVNS(x, k_max, t_max):
2      t ← 0
3      while t < t_max do
4          k ← 1
5          while k < k_max do
6              x' ← Shake(x, k)
7              x'' ← LocalSearch(x')
8              if IsBetter(x'', x) then
9                  x ← x''
10                 k ← 1
11             else
12                 k ← k + 1
13         end
14         t ← CPUTime()
15     end
16 return x
```

In Algorithm 6, we present the pseudocode of a VND procedure. This method receives two parameters: an initial solution x and a set of neighborhood structures N. First, the variable l is initialized to 1 (step 2). This variable indicates the next neighborhood to explore within the method. Then, at each iteration, a solution x' is obtained by performing a local search in the neighborhood structure N_l and starting with the solution x (step 4). If the resulting solution x', a local optimum within N_l, is better than the incumbent solution x (step 5), then x' is saved as the best solution (step 6) and l is reset to 1 (step 7) in order to restart the search from the first neighborhood structure in N. Otherwise, the value of l increases by 1 (step 9). The process stops when the best solution x is a local optimum within every neighborhood structure in N (step 3). At that point, the best solution found, x, is returned.

Algorithm 6: Pseudocode of the VND

```
1  Procedure VND(x, N):
2      l ← 1
3      while l ≤ |N| do
4          x′ ← LocalSearch(x, N_l)
5          if IsBetter(x′, x) then
6              x ← x′
7              l ← 1
8          else
9              l ← l + 1
10     end
11 return x
```

The pseudocode of GVNS is shown in Algorithm 7. The procedure receives four parameters: an initial solution x, a set of neighborhoods N, a maximum perturbation size k_{max}, and a maximum computation time t_{max}. First, a variable t is initialized to control the execution time of the method (step 2). Once the maximum amount of computing time has been consumed, the search is stopped (step 3). During the search, a variable k is first initialized to zero (step 4). This variable controls the size of the perturbation to be performed. At each iteration, a solution x' is obtained by perturbing the best solution x within a shake method (step 6). Then, the obtained solution x' is improved in a VND procedure (step 7). If the resulting solution x'' is better than the incumbent solution x (step 8), then x'' is saved as the best solution (step 9) and k is reset to zero (step 10). Otherwise, k is incremented to perform a larger perturbation and diversify the search process (step 12).

In addition to the different schemes mentioned above, one might find other extensions in the literature such as: Variable neighborhood formulation space search [101], which switches between different formulations of the same problem; variable formulation search, which switches between alternative objective functions; or parallel VNS [48], where different strategies are proposed to introduce parallelization in the VNS schemes. In addition, MO-VNS [39] extends the ideas of VNS for MOPs. This extension is discussed in the next section. For more information on VNS, its ideas, applications, and extensions, we refer the reader to relevant sources [50, 91, 100, 110].

Algorithm 7: Pseudocode of the GVNS method

```
1  Procedure GVNS(x, N, k_max, t_max):
2      t ← CPUTime()
3      while CPUTime() − t < t_max do
4          k ← 0
5          while k < k_max do
6              x′ ← Shake(x, k)
7              x″ ← VND(x′, N)
8              if IsBetter(x″, x) then
9                  x ← x″
10                 k ← 0
11             else
12                 k ← k + 1
13         end
14     end
15 return x
```

4.1.3 Multi-Objective Variable Neighborhood Search

Due to the competitive results achieved by VNS-based proposals in many domains, different extensions of the basic ideas behind the methodology have been investigated in the literature. In 2015, Duarte et al. proposed an extension of VNS for MOPs named MO-VNS [39]. To adapt VNS to the multi-objective context, MO-VNS first redefines the concept of a solution in the context of VNS. In MO-VNS, a solution is defined as an approximate set of efficient points found during the search process [39]. Accordingly, an efficient point is a particular vector of decision variables for the problem at hand (i.e., what is normally considered a solution for a problem in the context of VNS). Then, a solution is said to improve when a new efficient point is included in the set. Usually, an efficient point is included in the solution if it is not dominated by any point in the solution, although other inclusion criteria might be implemented.

In Algorithm 8, we present the pseudocode of the `MO-Improvement` method. This method receives two solutions: E and E'. Note that, in this context, a solution is a set of efficient points. Therefore, E and E' are sets of efficient points or Pareto fronts. Then, the method checks if there exists at least one efficient point in E' that is neither contained

Algorithm 8: Pseudocode of MO-Improvement

```
1 Procedure MO-Improvement(E,E'):
2     forall x ∈ E do
3         if x ∉ E ∧ ¬Dominated(x,E) then
4             return True
5     end
6 return False
```

nor dominated in E. If such an efficient point exists in E', then an improvement has been achieved. As can be observed, the method assumes that there are no efficient points in E that are not contained or dominated in E'. In the different MO-VNS schemes, E is always the incumbent solution, while E' is a solution that has been obtained after trying to improve E. Therefore, in the worst-case scenario, no improvement has been achieved and $E = E'$. If an improvement has been obtained, then the best solution found during the search process (E) is updated. That is, all efficient points in E' that are not contained in E are added to the solution E. Then, every dominated efficient point in E is removed.

Once the concept of a solution has been redefined within the MO-VNS framework, the well-known schemes and extensions of VNS can easily be adapted to the study of MOPs. For the remainder of this doctoral thesis, we will use the aforementioned terminology to describe the behavior of MO-VNS methods.

In Algorithm 9, we present the pseudocode of the MO-GVNS procedure. This method receives five parameters: a solution E, a set of neighborhood structures N, a maximum perturbation size k_{max}, a set of objectives R, and a maximum computing time t_{max}. The search process is performed until the maximum time t_{max} is reached (step 3). First, the variable k is initialized and set to one (step 4). Then the solution E is iteratively improved (steps 4- 13). At each iteration, three phases are performed. First, the efficient points in the solution E are perturbed by a `MO-Shake` procedure (step 6), which is detailed in Section 4.4.2. Then, the resulting solution, E', is improved in a `MO-VND` procedure (step 7). The procedure `MO-VND` improves each efficient point in the solution considering each objective $r \in R$ separately. If the resulting solution E'' contains any efficient point that is not dominated within E (step 8), then E is updated to include all non-dominated efficient points in $E \cup E''$ (step 9). Moreover, since the solution has been improved, k is reset to one

Algorithm 9: Pseudocode of the MO-GVNS method

```
1  Procedure MO-GVNS(E, N, k_max, R, t_max):
2  |  t ← CPUTime()
3  |  while CPUTime() − t < t_max do
4  |  |  k ← 1
5  |  |  while k < k_max do
6  |  |  |  E′ ← MO-Shake(E, k)
7  |  |  |  E″ ← MO-VND(E′, N, R)
8  |  |  |  if MO-Improvement(E, E″) then
9  |  |  |  |  E ← MO-Update(E, E″)
10 |  |  |  |  k ← 1
11 |  |  |  else
12 |  |  |  |  k ← k + 1
13 |  |  end
14 |  end
15 return E
```

Algorithm 10: Pseudocode of the MO-Shake procedure

```
1  Procedure MO-Shake(E, k):
2  |  E′ ← ∅
3  |  forall x ∈ E do
4  |  |  x′ ← Shake(x, k)
5  |  |  E′ ← E′ ∪ {x′}
6  |  end
7  return E′
```

(step 10). On the other hand, if no improvement has been achieved, the value of k increases by one (step 12). Finally, the solution E is returned.

In Algorithm 10, we present the pseudocode of the MO-Shake procedure. As can be observed, this method receives two parameters as input: a solution E and a perturbation size k. A new solution E' is first initialized (step 2). Then, each efficient point in the solution is perturbed by a shake method (step 4) and added to the new solution E' (step 5). After iterating through the entire set of efficient points in E (step 3), the new solution E' is returned.

Finally, in Algorithm 11, we present the pseudocode of the MO-VND method. This

Algorithm 11: Pseudocode of the MO-VND procedure

```
1  Procedure MO-VND(SE, N, R):
2      S_1 ← ∅, S_2 ← ∅, ..., S_|R| ← ∅
3      i ← 1
4      while i ≤ |R| do
5          while |SE \ S_i| ≥ 1 do
6              x ← Pick(SE \ S_i)
7              SE_i ← VND-i(x, N)
8              S_i ← S_i ∪ SE_i ∪ {x}
9          end
10         if MO-Improvement(SE, S_i) then
11             SE ← Update(SE, S_i)
12             i ← 1
13         else
14             i ← i + 1
15     end
16 return SE
```

method receives three input parameters: a solution SE, a set of neighborhood structures N, and a set of objectives R. First, the method initializes some sets of efficient points (step 2). Each set S_i will contain the efficient points that have already been visited considering the objective R_i. Then, the method performs a search process considering each objective separately (step 4). For each objective R_i, the entire set of efficient points in SE is explored considering only that objective (step 5). Those efficient points that have already been explored considering the objective Ri (i.e., those contained in S_i) are ignored (step 6). Then, the neighborhood structures of each efficient point $x \in SE \setminus S_i$ are explored within a VND-i method (step 7). This VND-i method explores the set of neighborhoods N of an efficient point x considering only one objective R_i as a traditional VND approach. However, VND-i has the particularity that, instead of returning the local optimum found, it returns the set of efficient points found during the search process (including the local optimum). Then, this set of efficient points and the incumbent efficient point x are added to the set of visited points S_i (step 8). When every efficient point in the solution has been explored, the procedure checks if an improvement has been made (step 10). If made, then the non-dominated efficient points found are added to the solution (step 11) and i is reset to one

(step 12). Otherwise, i is incremented and the search process continues considering the next objective in R (step 14). Once all objectives have been considered without improving the current solution, the method returns the set of non-dominated efficient points SE.

4.2 Algorithmic proposal for the MQ problem

For the MQ problem, this doctoral thesis proposed a method that hybridizes the GRASP and VNS methodologies. In particular, the local search phase of GRASP is replaced by a VND component. In Algorithm 12, we present the pseudocode of the algorithm proposed. As can be seen, the method receives two parameters: the MDG to be modularized (G) and the maximum number of iterations to be performed (*MaxIter*). The procedure starts with a preprocessing phase (step 2). This phase reduces the size of a given MDG in order to accelerate the following phases (the preprocessing phase is described in Section 4.2.1). Then, an initial solution x is built in the GRASP construction phase, considering the reduced MDG (step 6). Next, the initial solution x is improved using a VND method (step 7). If the resulting solution x' is better than the best solution found during the search, it is saved as the new best solution (steps 8-9). Finally, the best solution found during the search is returned.

Algorithm 12: Pseudocode of GRASP-VND

```
1  Procedure GRASP-VND(G, MaxIter):
2      G' ← PreprocessingPhase(G)
3      bestSolution ← ∅
4      i ← 0
5      while i < MaxIter do
6          x ← ConstructionPhase(G')
7          x' ← VND(x)
8          if IsBetter(x', bestSolution) then
9              bestSolution ← x'
10     end
11 return bestSolution
```

In the following sections, we describe each of the three phases included in the proposed GRASP-VND method. In particular, we detail the preprocessing reduction phase in Section 4.2.1, the constructive phase in Section 4.2.2, and the VND method in Section 4.2.3.

Additionally, the search parameter that determines the maximum number of iterations is experimentally set in Section 5.2.

4.2.1 Preprocessing reduction phase

In 2013, Köhler et al. proved the following theorem: "*Let $G = (V, E)$ be the undirected weighted graph given as input for the SCP. Let $u \in V$ be a node with degree equal to one and $v \in V$ be adjacent to u. Then in the optimal solution of the SCP, u and v are assigned to the same cluster*" [75]. Based on this theorem, the authors proposed a preprocessing procedure to reduce the size of a given MDG. The pseudocode of the procedure is shown in Algorithm 13. As can be observed, the procedure receives an undirected weighted graph $G = (V, E, W)$ as input. First, a copy of the graph (G') is obtained (step 2). Then, the algorithm iterates through the vertices in V' that have a degree equal to one (step 3). For each vertex u with $deg(u) = 1$, its adjacent vertex v is obtained (step 4) and a self-loop (v, v) is added to the new graph (step 5). If the self-loop (v, v) already exists in W', then the weight is increased by adding the weight of the edge (u, v) (step 7). Otherwise, the same weight that the edge between u and v has is assigned (steps 9 to 10). At the end of each iteration, u is removed from the new graph G', along with its associated edge (steps 11 to 13). Finally, the procedure returns the resulting graph $G' = (V', E', W')$, with $|V'| \leq |V|$.

As can be noticed, the procedure described expects an undirected weighted graph as input. However, it might be the case that the MDG at hand is not an undirected weighted graph. Nevertheless, a directed unweighted graph can be easily transformed into a valid graph for the preprocessing step. First, all edges are assigned an equal weight (e.g., one). Then, all edges connecting the same pair of vertices, regardless of the direction of the edge, are combined into a single undirected edge. The weight of the resulting edge is equal to the sum of the weights of the combined original edges.

In Figure 4.3, we illustrate the preprocessing procedure with an example. First, a directed unweighted graph is depicted in Figure 4.3(a). This MDG is transformed into an undirected weighted graph following the steps mentioned above. That is: (i) the edges are assigned a weight equal to one, (ii) the directions of the edges are removed, and (iii) edges with the same pair of nodes are combined. As a result, the solution shown in Figure 4.3(b)

Algorithm 13: Pseudocode of the preprocessing phase

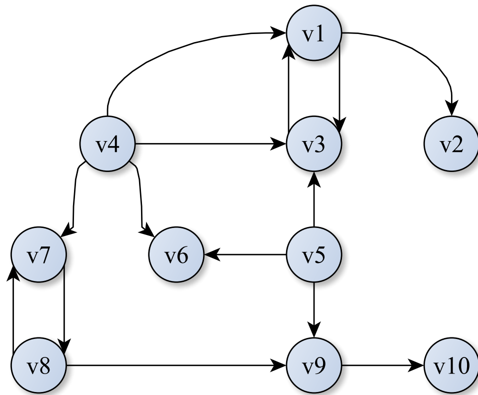
```
1  Procedure PreprocessingPhase(G):
2      G'(V',E',W') ← G(V,E,W)
3      for u ∈ V' : deg(u) = 1 do
4          v ← GetAdjacentVertex(G',u)
5          E' ← E' ∪ (v,v)
6          if w'_(v,v) ∈ W' then
7              w'_(v,v) = w'_(v,v) + w'_(u,v)
8          else
9              w'_(v,v) = w'_(u,v)
10             W' ← W' ∪ w'_(v,v)
11         V' ← V' \ u
12         E' ← E' \ (u,v)
13         W' ← W' \ w'_(u,v)
14     end
15 return G'
```

is obtained. Then, the graph is reduced by applying the method presented in Algorithm 13. First, $v2$ is removed from the graph, and a self-loop is added to $v1$. Since the weight of the edge $(v1, v2)$ is equal to one and the edge $(v1, v1)$ did not exist before, the weight of the resulting self-loop is equal to one, as depicted in Figure 4.3(c). The procedure continues by removing the vertex $v10$ and adding a self-loop $(v9, v9)$ with a weight equal to one, as shown in Figure 4.3(d). As can be observed, after the preprocessing procedure is applied, the resulting graph contains two vertices less than before ($v2$ and $v10$).

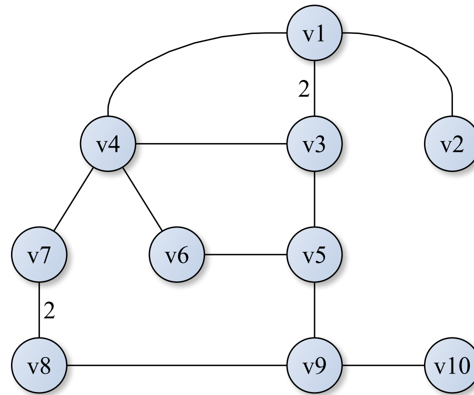
It is worth mentioning that the reduced graph obtained and the original MDG have the same TurboMQ value [75]. Once the search process has ended and a solution is found, the structure of the original MDG can be trivially recovered by adding back the removed vertices and placing them in the same module to which their adjacent vertices belong.

4.2.2 Constructive phase

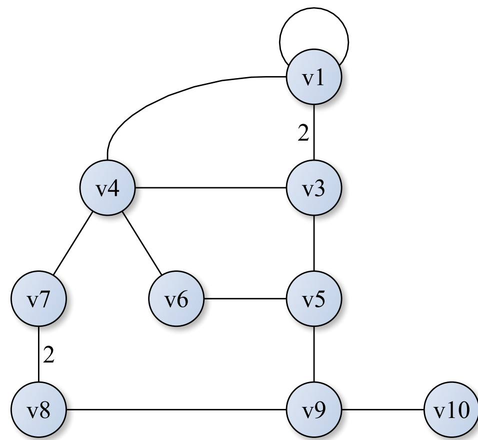
The constructive phase, introduced in Algorithm 12, starts with an empty solution (a solution with no vertices). In an iterative process, the method selects one vertex per iteration



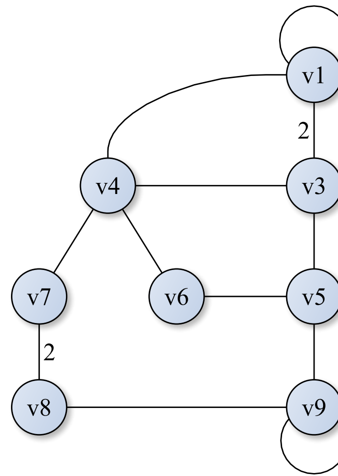
(a) Original directed unweighted MDG.



(b) Adapted undirected weigthed graph.



(c) Preprocessing procedure: removed $v2$.



(d) Preprocessing procedure: removed $v10$.

Figure 4.3 Resulting graph after the preprocessing phase. In Figure 4.3(a), the original MDG is depicted. In Figure 4.3(b), the original MDG is transformed into an undirected weighted graph. In Figure 4.3(c), vertices with a degree equal to one have been removed and a self-loop to $v1$ has been added. In Figure 4.3(d), vertex $v10$ has been removed and a self-loop to $v9$ has been added.

and adds it to the partial solution. The process ends when the solution is complete. That is, when every vertex has been added to the solution.

Formally, we define a partial solution as $S = \{S_1, S_2, ..., S_p\}$, where each S_i (with $1 \leq i \leq p$) represents the set of vertices belonging to module m_i. Accordingly, the set of vertices that are not included in the solution is defined as $Q = V \setminus \bigcup_{i=1}^{p} S_i$. Then, a solution S is partial iff $|Q| > 0$. Otherwise, if $\sum_{i=1}^{p} |S_i| = |V|$, then the solution is considered complete.

In Algorithm 14, we present the pseudocode of the constructive method. As can be observed, this pseudocode is very similar to the one presented in Algorithm 4. The adjustments introduced to adapt this method for the MQ problem are highlighted in green, in steps 6 and 9. As highlighted, the greedy function must be defined for the problem at hand. In addition, the adding function is now performed by a method named `AddVertex`, which will be described next.

Algorithm 14: Pseudocode of the constructive procedure

```
1  Procedure ConstructivePhase(G(V,E,W)):
2      x ← ∅
3      CL ← V
4      α ← Random(0,1)
5      while CL ≠ ∅ do
6          th ← min( g(v) ) + α · (max( g(v) ) − min( g(v) )) | v ∈ CL
7          RCL ← v ∈ CL | g(v) ≥ th
8          vertex ← RandomChoice(RCL)
9          x ← AddVertex(x, vertex)
10         CL ← CL \ vertex
11     end
12 return x
```

First, a vertex is selected at each iteration to be added to the solution, using a greedy function g. As mentioned in Section 4.1.1, the greedy function of a GRASP constructive procedure is problem dependent. In this case, the greedy function is designed to evaluate the "proximity" of each candidate vertex to the partial solution. The greedy criterion, for any given vertex v, is computed as a trade-off between the number of adjacent vertices to v that are already included in the partial solution and the number of adjacent vertices to v not yet included in the partial solution. In particular, the greedy function is defined as follows:

$$g(v) = \underbrace{\max_{S_i \,\in\, S} \left(\sum_{u \,\in\, S_i} w_{(v,u)}, 1 \leq i \leq k \right)}_{\text{(a)}} - \underbrace{\sum_{u \,\in\, Q} w_{(v,u)}}_{\text{(b)}}. \tag{4.1}$$

For each vertex v, the greedy function calculates: (a) the maximum sum of the weights of the edges connecting v with any vertex belonging to any module $S_i \in S$; and (b) the sum of the weights of the edges connecting v with any vertex $u \in Q$ not yet included in the partial solution. Then, it returns the difference between (a) and (b). As can be observed, the higher the number of adjacent vertices to v that are already included in the solution, the higher the chance that v is added to the solution.

In many problems, adding the selected candidate to the solution is a trivial task. However, for problems of the SMCP family, this task is not trivial, since a given vertex can be added to any module of the solution. In this case, once a candidate vertex v has been selected, the `AddVertex` method introduced in step 9 of Algorithm 14 (highlighted in green) is responsible for selecting the module where v is inserted. The `AddVertex` method evaluates the quality of the resulting solution considering the insertion of v into each existing module $S_i \in S$ and also into a new empty module. Moreover, since a module with a single vertex does not have cohesion, the method additionally considers the creation of a new module that contains both v and any vertex $u \in S$ that is adjacent to v. This promotes the creation of new modules in the partial solution. Finally, the decision that results in the solution with better quality is selected.

4.2.3 Variable Neighborhood Descent

The improvement phase of search algorithms aims to refine an initial solution until a local optimum is found. In the algorithmic proposal presented in Algorithm 12, the improvement phase, usually performed by a local search procedure, is replaced by a VND method, which is one of the most widely used extensions of VNS. In VND, a set of different neighborhoods is systematically explored. In Algorithm 15, we present the pseudocode of the VND component used in Algorithm 12. As can be observed, this method is similar to the procedure described in Algorithm 6. Some adjustments are introduced to adapt the local

Algorithm 15: Pseudocode of the VND component of the method proposed for the MQ problem.

```
1  Procedure VND (x, N = {N1, N3, N4}):
2      l ← 1
3      while l ≤ |N| do
4          switch l do
5              case 1
6                  x' ← LocalSearchFirstImprovement(x, N1)
7              end
8              case 2
9                  x' ← LocalSearchFirstImprovement(x, N3)
10             end
11             case 3
12                 x' ← LocalSearchFirstImprovement(x, N4)
13             end
14         end
15         if IsBetter(x', x) then
16             x ← x'
17             l ← 1
18         else
19             l ← l + 1
20     end
21 return x
```

search strategies and the neighborhood structures explored in steps 6, 9, and 12, which have been highlighted in green color. The set of neighborhood structures N contains three neighborhoods: N_1, N_3, and N_4. Since all procedures proposed in this doctoral thesis contain methods based on the exploration of neighborhood structures, all neighborhoods are described together in Section 4.5. In this case, the VND component of the algorithm proposed for the MQ problem explores the neighborhoods N_1, N_3, and N_4. The preliminary experiments devoted to the analysis of these neighborhoods and the order in which they should be explored are described later in Section 5.2. Additionally, it should be noted that the local search procedures described in the VND component of this proposal follow a First Improvement strategy. That is, when exploring the different neighborhood structures, the first move found in the neighborhood that improves the quality of the solution is applied.

4.3 Algorithmic proposal for the FCB problem

In this section, we describe a procedure based on GVNS, which is proposed for the FCB problem. Within the VNS methodology, several schemes have been proposed that implement the ideas of VNS in both stochastic and deterministic components. BVNS, which was presented in Algorithm 5, was the first scheme proposed in the methodology, implementing the ideas in a stochastic component, the perturbation phase, designed to escape local optima. On the contrary, VND, which was presented in Algorithm 6, does not include a perturbation phase, but rather explores a set of different neighborhood structures in a systematic way. GVNS, in contrast, combines a stochastic and a deterministic exploration of different neighborhood structures. The stochastic behavior is achieved by a perturbation phase, which modifies a given solution by performing random moves within a neighborhood structure to introduce diversification in the search, while the improvement phase is performed by a VND procedure.

The pseudocode of the GVNS method is described in Algorithm 7. The constructive procedure used to build the initial solution is described in Section 4.3.1. The VND component of the method is described in Section 4.3.2.

4.3.1 Constructive procedure

In this case, the initial solution x received as input in the GVNS method is built at random. The pseudocode of the constructive procedure is presented in Algorithm 16. The constructive procedure receives an MDG ($G(V, E, W)$) as input. First, an initial empty solution x is created (step 2). Then, $|V|$ modules are created, which are initially empty, and added to the solution x (steps 3-7). Next, each vertex of the graph is assigned to a random module, following a uniform distribution (steps 9-11). Finally, once every vertex belongs to a module in the solution, the modules that do not contain any vertices are removed (steps 13-15). The resulting solution x is then returned.

Algorithm 16: Pseudocode of the constructive procedure used in the method proposed for the FCB problem.

```
1  Procedure Constructive(G(V,E,W)):
2      x ← ∅
3      i ← 0
4      for u ∈ V do
5          S_i ← {}
6          x ← x ∪ S_i
7          i ← i + 1
8      end
9      for u ∈ V do
10         i ← Random(0,|V| − 1)
11         S_i ← S_i ∪ u
12     end
13     for S_i ∈ x do
14         if |S_i| = 0 then
15             x ← x \ S_i
16     end
17 return x
```

4.3.2 Variable Neighborhood Descent

In Algorithm 17, we present the pseudocode of the VND component used in the GVNS method proposed for the FCB problem. As can be observed, this method is similar to the procedure described in Algorithm 6. The difference lies in the local search strategies and the neighborhood structures explored in steps 6, 9, and 12, which have been highlighted in green color. The set of neighborhood structures N contains three neighborhoods: N_4, N_3, and N_1. Since all procedures proposed in this doctoral thesis contain methods based on the exploration of neighborhood structures, all neighborhoods are described together in Section 4.5. In this case, the VND component of the algorithm proposed for the FCB problem explores the neighborhoods N_4, N_3, and N_1. The preliminary experiments devoted to the analysis of these neighborhoods and the order in which they should be explored are described later in Section 5.2. Furthermore, it should be noted that the local search procedures described in the VND component of this proposal follow a First Improvement strategy. That is, when exploring the different neighborhood structures, the first move found

in the neighborhood that improves the quality of the solution is applied.

Algorithm 17: Pseudocode of the VND component of the method proposed for the FCB problem.

```
1  Procedure VND(x, N = {N_4, N_3, N_1}):
2      l ← 1
3      while l ≤ |N| do
4          switch l do
5              case 1
6                  x' ← LocalSearchFirstImprovement(x, N_4)
7              end
8              case 2
9                  x' ← LocalSearchFirstImprovement(x, N_3)
10             end
11             case 3
12                 x' ← LocalSearchFirstImprovement(x, N_1)
13             end
14         end
15         if IsBetter(x', x) then
16             x ← x'
17             l ← 1
18         else
19             l ← l + 1
20     end
21 return x
```

4.4 Algorithmic proposal for the MCA and ECA problems

In this section, we describe a procedure based on MO-GVNS, which is proposed for the MCA and ECA problems. As introduced in Section 2.3 and Section 2.4, the MCA and ECA problems share four quality metrics. For this reason, they are usually studied together in the literature. In this doctoral thesis, we propose a procedure based on MO-GVNS for both problems.

As described in Section 4.1.3, MO-GVNS is an adaptation of the well-known GVNS scheme for MOPs, based on the MO-VNS extension. Note that in this context, a solution is defined as a set of efficient points. That is, a Pareto front or a set of non-dominated efficient points.

As shown in Algorithm 9, the presented MO-GVNS algorithm receives an initial solution as input. In Section 4.4.1, we describe the construction procedure used to generate the initial set of efficient points. In the same way, the MO-Shake and MO-VND procedures are described in Sections 4.4.2 and 4.4.3, respectively.

4.4.1 Constructive procedure

To generate the initial solution for the MO-GVNS procedure presented in Section 4.4, we propose an agglomerative constructive based on the ideas of Path Relinking [52]. The constructive procedure starts by generating two trivial efficient points:

1. In the first efficient point, each vertex is contained in a different module. That is, there are as many modules as vertices in the efficient point, and all of them are isolated modules ($n = |V|$). This trivial efficient point is optimal in terms of the number of modules (considering MCA and ECA). However, it has zero cohesion and maximum coupling, which are not desirable properties of good modular organizations.

2. In the second efficient point, every vertex belongs to the same module. That is, the efficient point contains only one module ($n = 1$). This trivial efficient point has zero coupling and maximum cohesion, which are desirable properties. However, having only one module is not desirable.

Once these two trivial efficient points are generated, the procedure proceeds to build a path between them, obtaining one efficient point at each iteration. In particular, a solution is obtained at each iteration by merging two different modules into a single one.

In Figure 4.4, we illustrate this process with an example. As can be observed, the procedure starts with building an initial trivial efficient point x_s with five isolated modules (iteration 1). Then, all possible merges of two different modules are evaluated (iteration 2). Since both the MCA and ECA problems consider MQ as one of the objectives, we use the

MQ value to evaluate the quality of possible merges. The best possible merge in terms of MQ is selected and committed. Therefore, the resulting solution x' is obtained by merging the modules that contain vertices $v2$ and $v4$. Then, all possible merges are evaluated again (iteration 3). As a result, the solution x'' is obtained by merging the modules that contain vertices $v1$ and $v3$. The process continues iteratively until the second trivial efficient point x_t is obtained (iteration 5), where all vertices belong to the same module. Finally, all the efficient points generated during the process (x_s, x', x'', x''', and x_t) are added to the initial solution E.

In Figure 4.5, we represent a solution E obtained by the described constructive procedure. The efficient points are represented by a red dot in a two-dimensional objective space. The x-axis represents the MQ value, while the y-axis represents the number of modules in each efficient point. The constructive procedure generates efficient points along the objective space by iteratively reducing the number of modules by one while simultaneously improving the MQ value of the efficient point obtained at each iteration. As a result, the constructive procedure generates efficient points from the upper left corner to the lower right corner of the objective space represented in the figure, and none of the efficient points generated is dominated within the initial solution. As can be seen, all the generated efficient points are not dominated by the rest, since each obtained efficient point has less modules but better MQ value than the previous one.

In this case, the greedy criterion used to evaluate the possible merge operations at each iteration is TurboMQ. As illustrated in Figure 4.4, the merge operation selected at each iteration is the one that results in the solution with the maximum TurboMQ value among the candidates. However, the procedure can be trivially adapted to use any other criterion for the selection of the operation to perform at each iteration. Moreover, semi-greedy selection mechanisms can also be used to add some randomization to the process. Finally, it is worth noting that only the merges possible in the efficient point obtained after each iteration are evaluated. Thus, the whole search space is not explored.

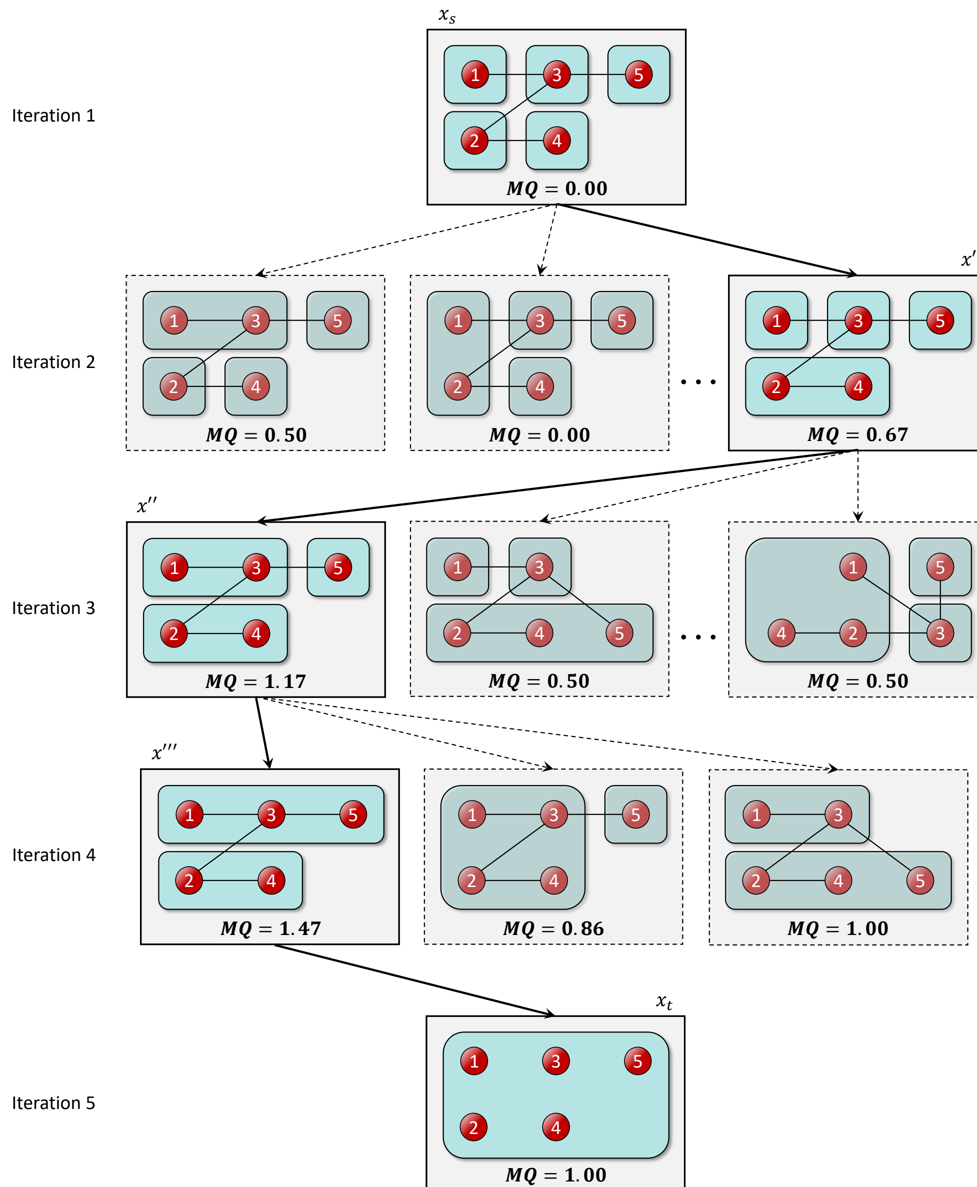


Figure 4.4 Example of the agglomerate constructive process for an MDG with five vertices. Five iterations are shown, where a path is built linking two trivial efficient points: x_s and x_t.

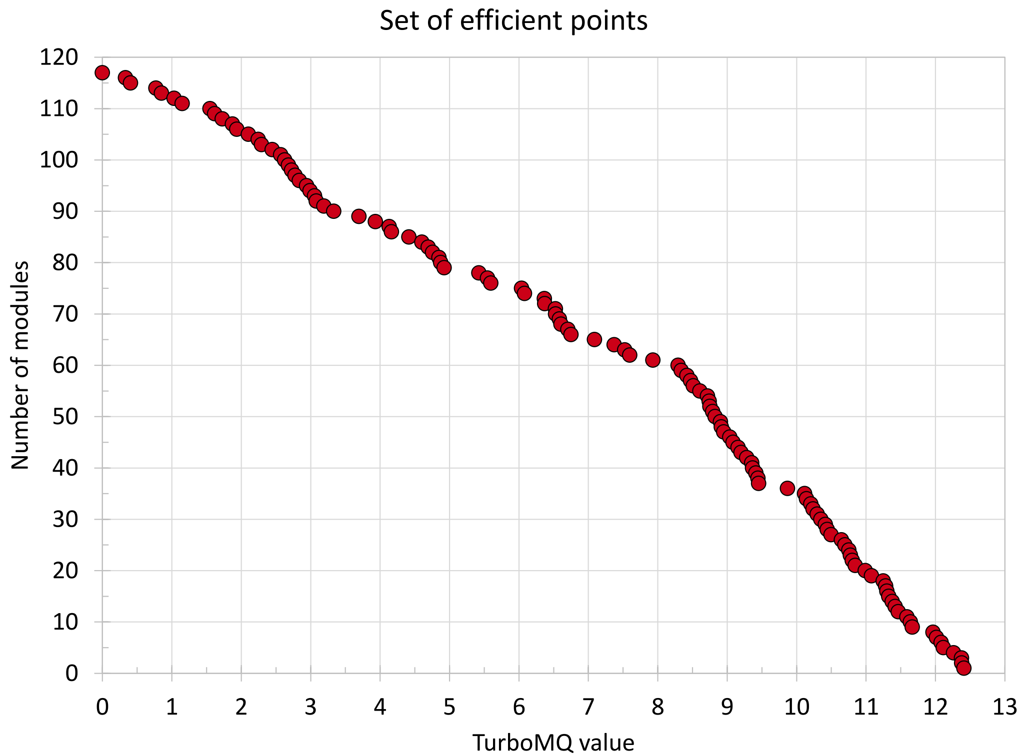


Figure 4.5 Representation of a solution generated by the MO-GVNS constructive procedure in the objective space of two quality metrics: MQ and the number of modules. Note that all efficient points depicted are not dominated, since both objectives should be maximized.

4.4.2 Multi-Objective Shake

In MO-GVNS, the MO-Shake component extends the idea of the shake method of GVNS, by performing a perturbation in each efficient point within the solution. For the shake procedure, a neighborhood structure is proposed based on swap operations. Here, a swap operation is based on interchanging the positions of two vertices. Given two vertices $v1$ and $v2$ that belong to modules m_1 and m_2, respectively, swapping them will result in vertex $v1$ belonging to module m_2 and vertex $v2$ belonging to module m_1. The number of swap operations to perform within the `Shake` procedure is determined by the value of k. The higher the value of k, the higher the number of swap operations to perform.

Here, we propose four different shake procedures based on the neighborhood mentioned

Table 4.1 Selection criteria used for each of the two vertices involved in a swap operation, for each of the proposed shake procedures.

	Selection criterion	
	First vertex	Second vertex
Shake 1	Random	Random
Shake 2	Greedy	Greedy
Shake 3	Random	Greedy
Shake 4	Greedy	Random

earlier. These procedures differ in the criteria used to select the vertices to swap. We propose two different selection criteria for the first vertex and two different selection criteria for the second vertex. In both cases, one criterion is random, and the other is greedy. In Table 4.1, we present the proposed criteria. As can be observed, the resulting shake procedures are purely random (Shake 1), purely greedy (Shake 2), or a combination of random and greedy criteria (Shakes 3 and 4). The shake procedures presented here are later empirically evaluated in Section 5.2.2. In particular, the proposed shakes and their selection criteria are designed as follows:

- **Shake 1**. In the first shake, both vertices are randomly selected from any module in the efficient point.
- **Shake 2**. In the second shake, both vertices are greedily selected. For the first vertex, we find the module m_i that has the worst CF value (see Equation 2.6). Then, a random vertex vi is selected within that module. For the second vertex, we select the vertex most closely related to module m_i (calculated as the sum of the weights of the edges that connect a given vertex to vertices in module m_i).
- **Shake 3**. In the third shake, the first vertex is selected at random. Then, the second vertex is selected as described in Shake 2. That is, for the second vertex, we select the one with the strongest connection to the module of the first vertex selected.
- **Shake 4**. In the fourth shake, we first find the module m_i that has the worst CF value within the efficient point (see Equation 2.6). Then, the first vertex is selected at random from that module. The second vertex is selected at random.

Algorithm 18: Pseudocode of the MO-VND component of the method proposed for the MCA and ECA problems

```
1  Procedure
     MO-VND(SE, N = {N1, N3, N2, N4}, R = {MQ, Cohesion, NumberOfModules}):
2      S1 ← ∅, S2 ← ∅, ..., S|R| ← ∅
3      i ← 1
4      while i ≤ |R| do
5          while |SE \ Si| ≥ 1 do
6              x ← Pick(SE \ Si)
7              switch i do
8                  case 1
9                      SEi ← VND-MQ(x, N = {N1, N3, N2, N4})
10                 end
11                 case 2
12                     SEi ← VND-Cohesion(x, N = {N1, N3, N2, N4})
13                 end
14                 case 3
15                     SEi ← VND-NumberOfModules(x, N = {N1, N3, N2, N4})
16                 end
17             end
18             Si ← Si ∪ SEi ∪ {x}
19         end
20         if MO-Improvement(SE, Si) then
21             SE ← Update(SE, Si)
22             i ← 1
23         else
24             i ← i + 1
25     end
26 return SE
```

4.4.3 Multi-Objective Variable Neighborhood Descent

In Algorithm 18, we present the pseudocode of the MO-VND component of the MO-GVNS method proposed for the MCA and ECA problems. As can be observed, this method is similar to the procedure described in Algorithm 11. The difference lies in the the neighborhood structures explored and objectives considered. The differences in the pseudocode have been highlighted in green color. The set of neighborhood structures N contains four neighborhoods: N_1, N_3, N_2, and N_4. Since all procedures proposed in this doctoral thesis contain methods based on the exploration of neighborhood structures, all neighborhoods are described together in Section 4.5. The objectives considered in R are: MQ, cohesion, and the number of modules. The objectives to be explored within the method and the order in which they are explored are experimentally configured in Section 5.2.3. Furthermore, it should be noted that the local search procedures described in the MO-VND component of this proposal follow a First Improvement strategy. That is, when exploring the different neighborhood structures, the first move found in the neighborhood that improves the quality of the solution is applied.

4.5 Neighborhood structures

All procedures proposed in this doctoral thesis are based on the exploration of multiple neighborhood structures through the use of VNS. In this context, it is desirable that the neighborhoods included in the methods complement each other. As discussed previously, for the SMCP, a trade-off must be made between the number of modules and the size of each module (see Section 2). This balance has been promoted differently depending on the particular problem (e.g., MQ, FCB, MCA, or ECA). Nevertheless, it is undoubtedly important to allow a search algorithm to alter the number of modules in any solution. Here, we propose a categorization of neighborhood structures into three different groups, depending on the impact that they have on the number of modules:

1. Neighborhood structures defined by operations that do not alter the number of modules. The exploration of these neighborhood structures is intended to lead to the movement of vertices between existent modules.

2. Neighborhood structures defined by operations that increase the number of modules. The exploration of these neighborhood structures is intended to create new modules within the solution.

3. Neighborhood structures defined by operations that reduce the number of modules. The exploration of these neighborhood structures is intended to remove one or more modules from the solution, relocating the vertices in those modules into other existing modules.

Given the aforementioned classification, we propose six different neighborhood structures. Two of them (N_1 and N_2), described in Sections 4.5.1 and 4.5.1, belong to the first category; two of them (N_3 and N_6), described in Sections 4.5.2 and 4.5.2, belong to the second category; and two of them (N_4 and N_5), described in Sections 4.5.3 and 4.5.3, belong to the third category.

4.5.1 Neighborhood structures defined by operations that do not alter the number of modules

In this section, we define neighborhood structures defined by operations that do not alter the number of modules. In particular, two neighborhoods are proposed: N_1, based on an insert operator; and N_2, based on a swap operator. Both neighborhoods are described next.

N1: Insert

This neighborhood structure is based on insert operations. An insertion is a classic move in optimization research [28, 51, 108]. It consists of changing the position of a single component of the solution. In this case, an insertion is defined as the relocation of a given vertex to a module different from the one to which it belongs. In Figure 4.6, we present an example of this operation. In particular, we represent the insertion of vertex $v1$ into module m_1. On the left, we present the initial solution x, where the vertex $v1$ belongs to module m_2. On the right, we present the resulting solution x', where the vertex $v1$ is now contained in module m_1.

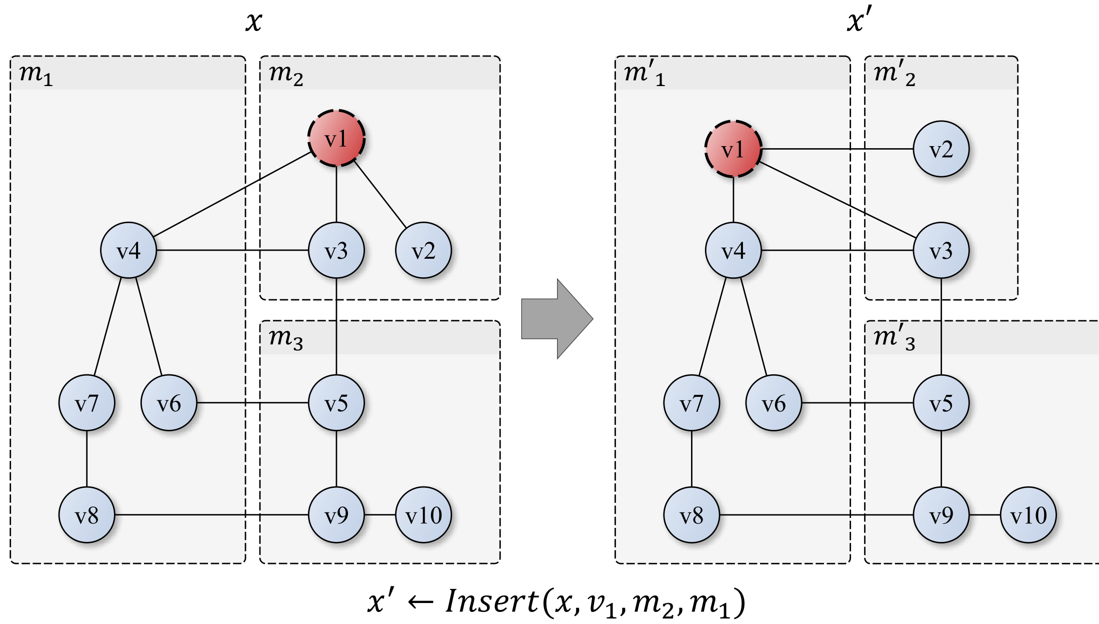


Figure 4.6 Example of an insert operation. Given the solution x on the left side, the solution x' on the right side is obtained by inserting vertex $v1$ into module m_1.

Formally, we define an insert operation as $x' \leftarrow Insert(x, v, m, m_t)$, where x is the solution at hand, v is a vertex contained in module m, m_t is the target module where v will be inserted, and x' is the resulting solution after applying the insert operation. Therefore, Figure 4.6 represents the operation $x' \leftarrow Insert(x, v_1, m_2, m_1)$.

Given the above definition of the insert operation, the resulting neighborhood, denoted as N_1, of any solution x, which is formed by all the solutions that can be obtained by applying any insert operation to x, is defined as follows:

$$N_1(x) = \{x' \leftarrow Insert(x, v, m, m_t) : \forall v \in V, \forall m_t \in M \; / \; v \in m, m \neq m_t\}.$$

According to the definition of the neighborhood, each combination of a pair (v, m_t) results in a different neighbor. Therefore, given a solution x with $|V|$ vertices and $|M|$ modules, the size of the neighborhood is $|V| \cdot (|M| - 1)$.

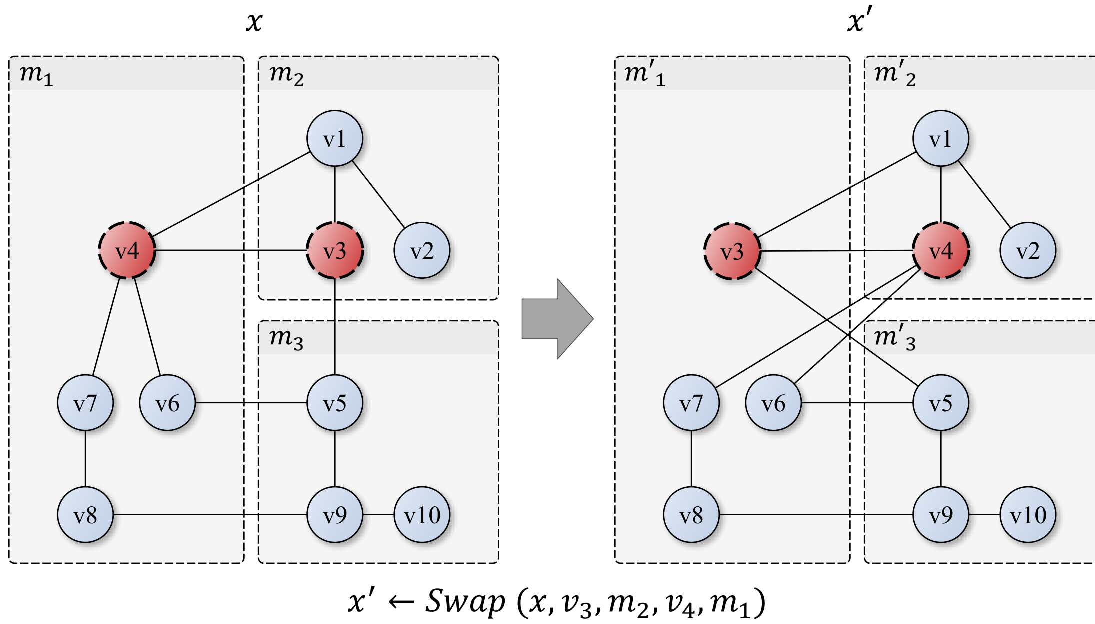


Figure 4.7 Example of a swap operation. Given the solution x on the left side, the solution x' on the right side is obtained by swapping vertices $v3$ and $v4$.

N2: Swap

The second neighborhood proposed is based on swap operations. Again, a swap operation is a classic move in optimization research [20, 49, 107]. It consists of interchanging the position of two different vertices. That is, given two vertices $v1$, contained in module m_1, and $v2$, contained in module m_2, swapping them would result in vertex $v1$ belonging to module m_2 and vertex $v2$ belonging to module m_1. In Figure 4.7, we present an example of this operation. In particular, we represent the swap of vertices $v3$ and $v4$. On the left, we present the initial solution x, where the vertex $v3$ belongs to module m_2 and the vertex $v4$ belongs to module m_1. On the right, we present the resulting solution x', where the vertex $v3$ belongs to module m_1' and the vertex $v4$ belongs to module m_2'.

Formally, we define a swap operation as $x' \leftarrow Swap(x, v_i, m_k, v_j, m_l)$, where x represents the initial solution, vi is a vertex that belongs to module m_k, vj is a vertex that belongs to module m_l, and x' is the resulting solution obtained after applying the swap operation. Therefore, in Figure 4.7, the operation represented is $x' \leftarrow Swap(x, v_3, m_2, v_4, m_1)$.

Given the above definition of the swap operation, the resulting neighborhood, denoted as N_2, of any solution x, which is formed by all the solutions that can be obtained by applying any swap operation to x, is defined as follows:

$$N_2(x) = \{x' \leftarrow Swap(x, v_i, m_k, v_j, m_l) : \forall v_i, v_j \in V \\ / \, v_i \in m_k \wedge v_j \in m_l \wedge m_k, m_l \in M \wedge m_k \neq m_l\},$$

where $1 \leq i, j \leq |V|$ and $1 \leq k, l \leq |M|$.

According to the definition of the neighborhood, given a solution x with $|V|$ vertices, the size of the neighborhood in the worst case is $|V| \cdot \frac{V-1}{2}$. In practice, the size of the neighborhood is smaller since the vertices within the same module cannot be swapped.

4.5.2 Neighborhood structures defined by operations that increase the number of modules

In this section, we define neighborhood structures defined by operations that increase the number of modules. In particular, two neighborhoods are proposed: N_3, based on a extract operator; and N_6, based on a split operator. Both neighborhoods are described next.

N3: Extract

The third neighborhood proposed is based on an extraction operation. This operation consists of inserting some vertices from one or more modules into a new empty module. In Figure 4.8, we present an example of this operation. In particular, we represent the extraction of vertices $v1$ and $v4$. On the left, we present the initial solution x, where the vertex $v1$ belongs to module m_2 and the vertex $v4$ belongs to module m_1. On the right, we present the resulting solution x', where the vertices $v1$ and $v4$ belong to a new module m'_4. Note that the resulting solution contains an additional module, m'_4.

Formally, we define an extraction operation as $x' \leftarrow Extract(x, O)$, where x represents the initial solution, O is a set of vertices, and x' is the solution that results after extracting all vertices in O to a new module. Therefore, in Figure 4.8, the operation represented is

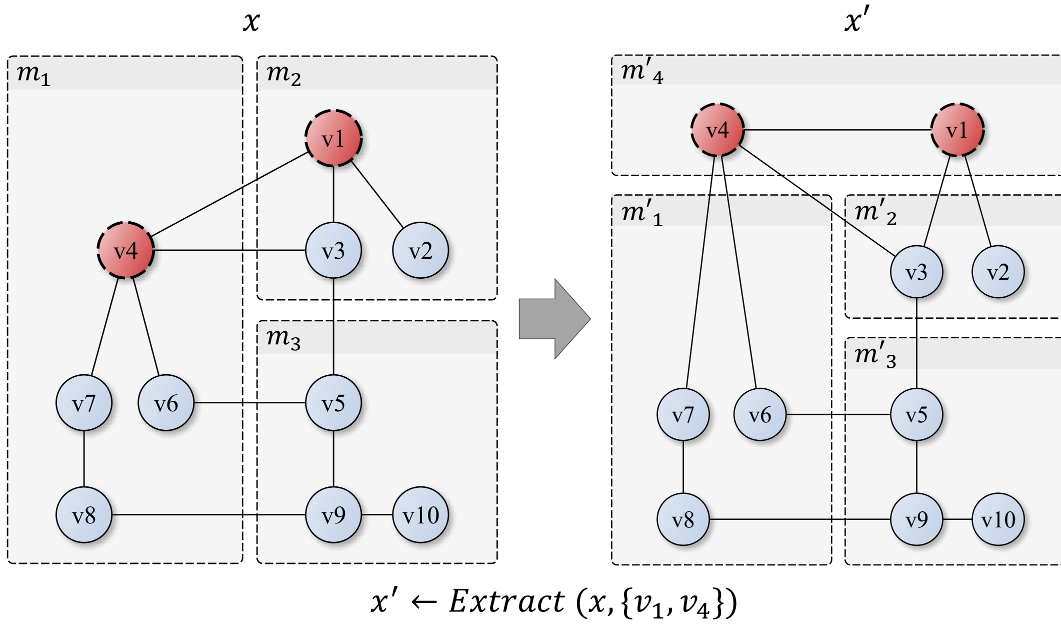


Figure 4.8 Example of a extract operation. Given the solution x on the left side, the solution x' on the right side is obtained by extracting vertices $v1$ and $v4$ to a new module m'_4.

$x' \leftarrow Extract(x, \{v_1, v_4\})$.

Given the above definition, the resulting neighborhood, denoted as N_3, of any solution x, which is formed by all the solutions that can be obtained by applying any extraction operation to x, is defined as follows:

$$N_3(x) = \{x' \leftarrow Extract(x, O) \ : \ O \subseteq V\}.$$

According to the above definition of the neighborhood, given a solution x with $|V|$ vertices, the size of the neighborhood in the worst case is $|V|!$. As can be noticed, the size of the neighborhood can become extremely large and unmanageable. Therefore, in practice, we limit the neighborhood to include only neighbors that can be obtained by performing extraction operations with only two or three vertices, as follows:

$$N_3(x) = \{x' \leftarrow Extract(x, O) \ : \ O \subseteq V \ / \ 2 \leq |O| \leq 3\}.$$

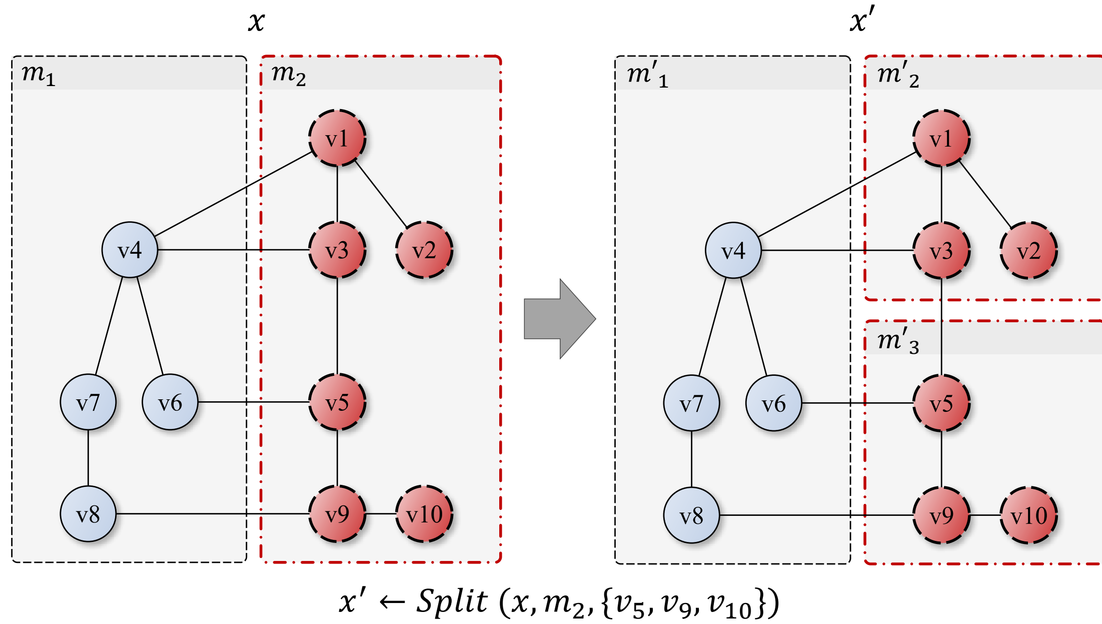


Figure 4.9 Example of a split operation. Given the solution x on the left side, the solution x' on the right side is obtained by splitting module m_2 into two new modules m'_2 and m'_3.

Therefore, the size of this neighborhood in practice is $V \cdot (V-1) + V \cdot (V-1) \cdot (V-2)$. It is worth mentioning that the size of this neighborhood structure can be adapted by changing the aforementioned limit depending on the problem studied.

N6: Split

The sixth proposed neighborhood is based on a split operation. This operation consists of dividing a module into halves. In Figure 4.9, we present an example of this operation. In particular, we represent the splitting of the module m_2. On the left, we present the initial solution x, where vertices $v1$, $v2$, $v3$, $v5$, $v9$, and $v10$ belong to module m_2. On the right, we present the resulting solution x', where module m_2 has been divided into the new modules m'_2 and m'_3. Notice that the resulting solution now contains one additional module and vertices $v1$, $v2$, and $v3$ belong to module m'_2, while vertices $v5$, $v9$, and $v10$ belong to module m'_3.

Formally, we define a split operation as $x' \leftarrow Split(x, m, O)$, where x represents the initial solution, m is a module to be split, and O is a set of vertices (with $\frac{|m|}{2} \approx |O|$) to be inserted into a new empty module. Therefore, in Figure 4.9, the operation represented is $x' \leftarrow Split(x, m_2, \{v_5, v_9, v_{10}\})$.

Given the above definition, the resulting neighborhood, denoted as N_6, of any solution x, which is formed by all the solutions that can be obtained by applying any split operation to x, is formally defined as follows:

$$N_6(x) = \{x' \leftarrow Split(x, m, O) : \forall\, m \in M \,/\, |O| = \lfloor \frac{|m|}{2} \rfloor \},$$

where M represents the set of modules in the solution x. If all possible distributions of the vertices in m are considered, the size of the entire neighborhood is equal to $\sum_{i=1}^{|M|} \frac{|m|!}{|O|!(|m|-|O|)!}$. As can be seen, the size of the entire neighborhood is prohibitively large. In practice, we bound it to find a balance between performance and time consumption. In particular, we only explore $|V|$ solutions. Each solution is constructed by dividing the vertices of a module considering their adjacency. In particular, we consider each vertex as a candidate seed. The seed is inserted into a new empty module together with half of the vertices from the same module that have the strongest dependencies towards the seed in terms of adjacency. The remaining vertices of the module are inserted into another new empty module. Since all vertices are considered as possible seeds, the size of this neighborhood is $|V|$.

4.5.3 Neighborhood structures defined by operations that reduce the number of modules

In this section, we define neighborhood structures defined by operations that reduce the number of modules. In particular, two neighborhoods are proposed: N_4, based on a destroy operator; and N_5, based on a merge operator. Both neighborhoods are described next.

N4: Destroy

The fourth proposed neighborhood is based on a destruction operation. This operation consists of removing a module from the solution and reinserting the vertices that belonged to

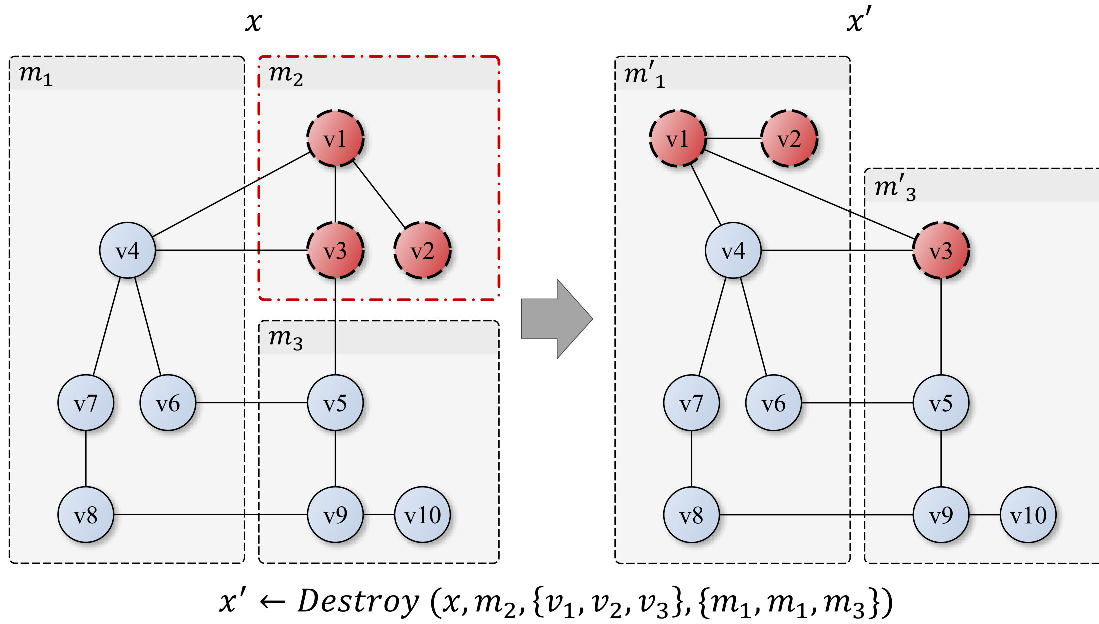


Figure 4.10 Example of a destroy operation. Given the solution x on the left side, the solution x' on the right side is obtained by destroying module m_2 and reinserting vertices $v1$, $v2$, and $v3$ into modules m_1', m_1', and m_3', respectively.

that module into other modules. In Figure 4.10, we present an example of this operation. In particular, we represent the destruction of module m_2. On the left, we present the initial solution x, where vertices $v1$, $v2$, and $v3$ belong to module m_2. On the right, we present the resulting solution x', where module m_2 has been destroyed and vertices $v1$, $v2$, and $v3$ have been relocated to modules m_1', m_1', and m_3', respectively. Note that the resulting solution contains one module less.

Formally, we define a destruction operation as $x' \leftarrow Destroy(x, m, O, D)$, where x represents the initial solution, m is a module to be removed, O is an ordered list of the vertices that belong to module m, D is an ordered list of modules where vertices contained in module m must be relocated to, and x' is the solution that results after the operation is applied. Notice that there exists an order correspondence between O and D, such that vertex O_i is inserted in module D_i (with $1 \leq i \leq |O|$). Therefore, in Figure 4.10, the operation represented is $x' \leftarrow Destroy(x, m_2, \{v_1, v_2, v_3\}, \{m_1, m_1, m_3\})$.

Given the above definition, the resulting neighborhood, denoted as N_4, of any solution x, which is formed by all the solutions that can be obtained by applying any destruction operation to x, is formally defined as follows:

$$N_4(x) = \{x' \leftarrow Destroy(x, m, O, D) \; : \; \forall m \in M \; / \; |O| = |D|, m \notin D\},$$

where O contains all vertices that belong to module m (and no more) and D is a list of modules where there can be repetitions, but not module m.

According to the above definition of the neighborhood, given a solution x with $|V|$ vertices and $|M|$ modules, the size of the neighborhood is $|M| \cdot (|M|-1)^{(|V|/|M|)}$, where $|V|/|M|$ is the average number of vertices per module. As can be noticed, the size of the resulting neighborhood structure can be very large. In practice, we bound it to find a balance between performance and time consumption. In particular, we do not consider all the possible insertions when a module is destroyed. Instead, each vertex is inserted only in the module that has the most dependencies towards that vertex. Therefore, in practice, the size of this neighborhood structure is equal to the number of modules $|M|$.

N5: Merge

The fifth proposed neighborhood is based on a merge operation. This operation consists of merging two modules into a single one. In Figure 4.11, we present an example of this operation. In particular, we represent the merging of modules m_2 and m_3. On the left, we present the initial solution x, where vertices $v1$, $v2$, and $v3$ belong to module m_2, while vertices $v5$, $v9$, and $v10$ belong to module m_3. On the right, we present the resulting solution x', where modules m_2 and m_3 have been merged into the new module m_2'. Notice that the resulting solution contains one module less and that vertices $v1$, $v2$, $v3$, $v5$, $v9$, and $v10$ now belong to the new module m_2'.

Formally, we define a merge operation as $x' \leftarrow Merge(x, m_i, m_j)$, where x represents the initial solution, and m_j is the module that will be merged into module m_i. That is, m_i will contain all the vertices in $m_i \cup m_j$. Therefore, in Figure 4.11, the operation represented is $x' \leftarrow Merge(x, m_2, m_3)$.

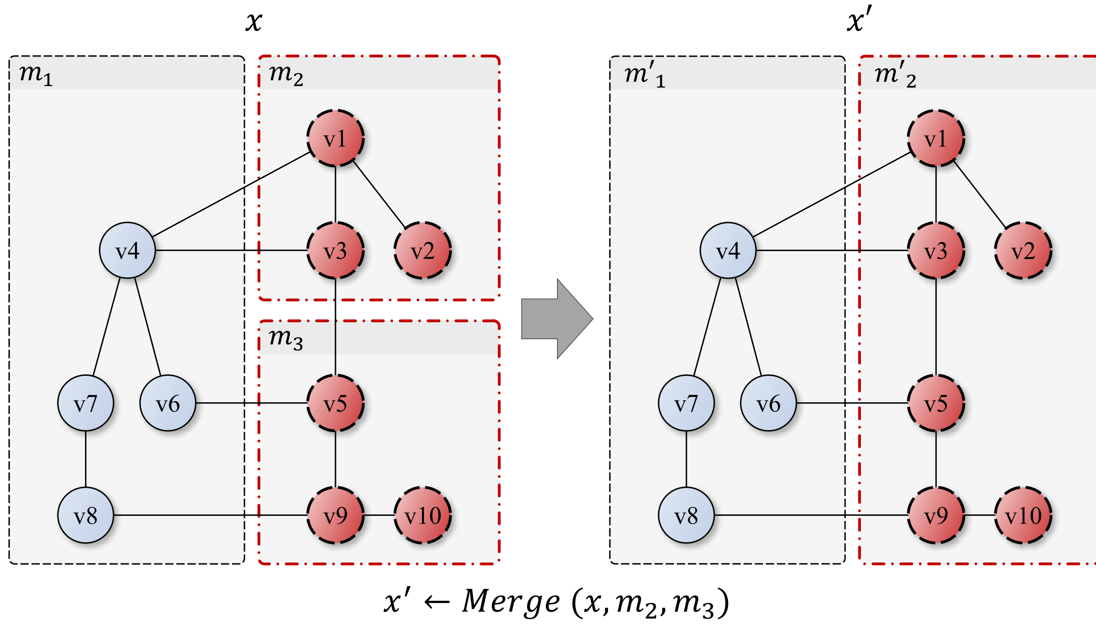


Figure 4.11 Example of a merge operation. Given the solution x on the left side, the solution x' on the right side is obtained by merging modules m_2 and m_3 into the new module m_2'.

Given the above definition, the resulting neighborhood, denoted as N_5, for any solution x, which is formed by all the solutions that can be obtained by applying any merge operation to x, is formally defined as follows:

$$N_5(x) = \{x' \leftarrow Merge(x, m_i, m_j) \; : \; \forall \, m_i, m_j \in M \, / \, i \neq j\},$$

where M represents the set of modules in x. According to the above definition of the neighborhood, given a solution x with $|M|$ modules, the size of the neighborhood is $|M| \cdot \frac{|M|-1}{2}$.

4.6 Advanced strategies

To improve the efficiency of the algorithmic proposals described in previous sections, here we introduce three different advanced strategies. First, in Section 4.6.1, we describe a strategy to accelerate the computation of the quality metrics. This strategy is applied to all the problems studied in this doctoral thesis: MQ, FCB, MCA, and ECA. Then, in Section 4.6.2, we introduce a strategy to reduce the size of the neighborhoods by exploring only promising regions in the search space. Again, this strategy is applied to all the problems studied in this doctoral thesis: MQ, FCB, MCA, and ECA. Finally, in Section 4.6.3, we describe a strategy to analyze the contribution of the guiding functions in a multi-objective context and reduce the set of guiding functions used during the search process. This strategy is applied in two of the problems studied in this doctoral thesis: MCA, and ECA.

4.6.1 Incremental evaluation of the objective functions

The first advanced strategy is designed to accelerate the computation of the objective functions during the search process. As previously described, the value of TurboMQ is calculated as the sum of different factors (see Equation 2.9). In particular, it is calculated as the sum of the cluster factor CF of each module in the solution. As can be observed in the calculation of the cluster factor of a module (see Equation 2.6), this value depends only on the vertices contained within that particular module. Therefore, as long as a module remains unaltered, its CF value does not change.

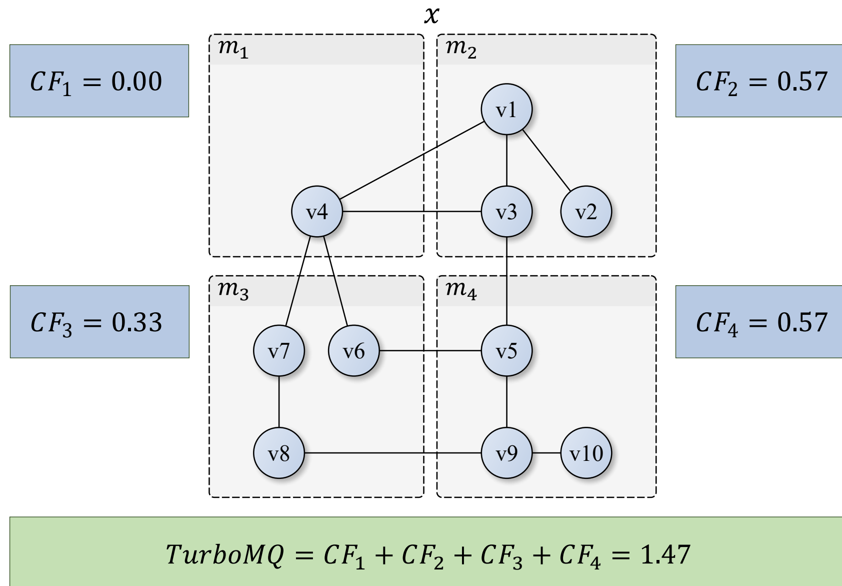
In trajectory-based search methods, a solution is usually modified over time by making changes to its decision variables. For example, in the case of the insertion neighborhood, a change in the solution involves the relocation of a single vertex from one module to another. Thus, an insert operation only changes the CF value of at most two modules. Therefore, if the CF value of the rest of the modules in the solution is already known, the TurboMQ value of the solution can be easily obtained by only recalculating the CF value of the modified modules.

In Figure 4.12, we illustrate this concept with an example. As can be observed in Figure 4.12(a), an initial solution x is presented, with four different modules. To obtain the TurboMQ value of the solution x, we first need to calculate the CF value of each module

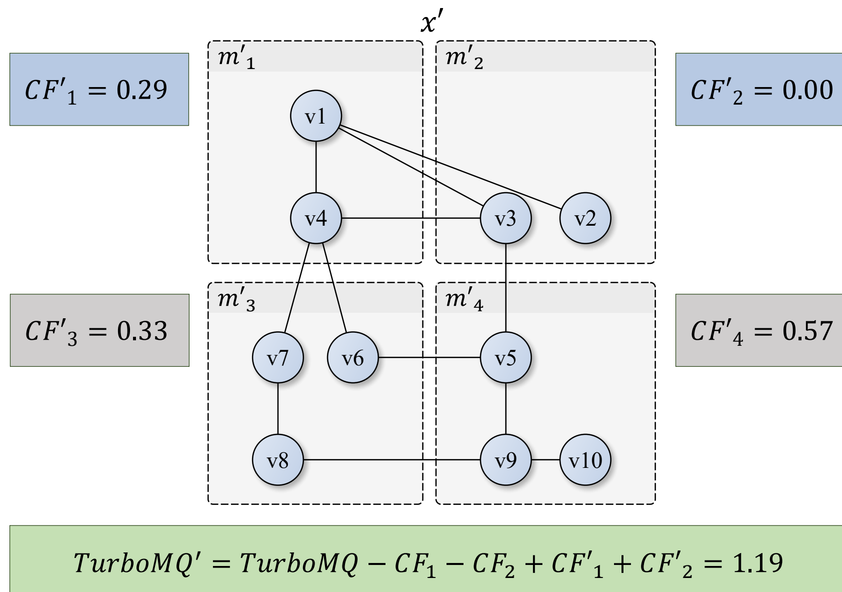
within the solution (presented in rectangular boxes filled with blue). The resulting value of TurboMQ is equal to 1.47 (presented in a rectangular box filled with green). Then, an insert operation is applied to the solution x, where the vertex $v1$ is relocated from module m_2 to module m_1. The resulting solution x' is presented in Figure 4.12(b). Once again, to obtain the TurboMQ value of the solution x', we first need to obtain the CF value of each module within the solution. However, this time, the CF values of modules m'_3 and m'_4 are already known (presented in rectangular boxes filled with gray) as they have not been modified. Therefore, we can obtain the value of the TurboMQ objective function by recalculating only the CF values of modules m'_1 and m'_2 (presented in rectangular boxes filled with blue). As can be observed, only two CF values have been calculated. Furthermore, the new TurboMQ value (*TurboMQ'*) can be calculated by updating the CF values of the affected modules, instead of adding the CF values of each module in the solution. The benefits of the incremental evaluation are not exclusively attained after performing insert operations, but after performing any operation. To illustrate it, in Figure 4.12(c) we present a third solution x'' obtained after performing a destruction operation ($x'' \leftarrow Destroy(x', m_4, \{v_5, v_9, v_{10}\}, \{m'_3, m'_3, m'_3\})$) to the previous solution x'. Accordingly, only the CF value of the module m''_3 is recalculated.

It should be noted that the number of CF values to recalculate depends solely on the number of modules that have been modified, not on the number of modules within the solution. After an insert operation, only two CF values must be recalculated, regardless of the size of the solution. Therefore, the cost of recomputing the TurboMQ value remains constant with respect to the size of the project. The larger the graph, the larger the benefit of the incremental evaluation.

This idea of incremental evaluation can easily be extended to the rest of the quality metrics studied in this doctoral thesis. In the case of FCB, the value of the objective function is obtained by adding the coupling of the entire architecture and the maximum cohesion of any module (see Equation 2.10). If the cohesion and coupling of each module are previously known and stored, only the cohesion and coupling of the modules that have been modified after each operation need to be recalculated. Then, the FCB value can be updated accordingly. In Figure 4.13 and Figure 4.14, we present the incremental evaluation of a solution after a swap operation. First, an initial solution x is presented in Figure 4.13. The



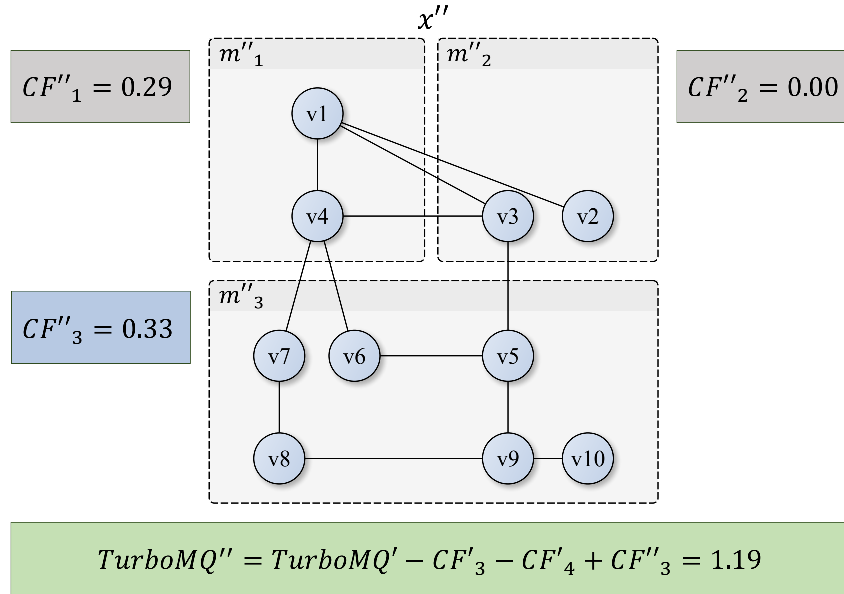
(a) Calculation of the TurboMQ value of a solution x.



(b) Incremental evaluation of the TurboMQ value of a solution x', obtained after applying the operation $x' \leftarrow Insert(x, v_1, m_2, m_1)$.

continued on the next page

Figure 4.12 Incremental evaluation of the TurboMQ value for a given solution after performing an insert operation and a destruction operation.



(c) Incremental evaluation of the TurboMQ value of a solution x'', obtained after applying the operation $x'' \leftarrow Destroy(x', m'_4, \{v_5, v_9, v_{10}\}, \{m'_3, m'_3, m'_3\})$.

Figure 4.12 Incremental evaluation of the TurboMQ value of a given solution after performing an insert operation and a destruction operation.

computation of the cohesion and coupling values for each module is shown in rectangular boxes filled with blue. The calculation of the FCB value is shown in a rectangular box filled with green. Then, in Figure 4.14, we present a solution x', obtained after performing the operation $x' \leftarrow Swap(x, v_3, v_4)$. As can be observed, only the coupling and cohesion values of two modules (m'_1 and m'_2) are recalculated (presented in rectangular boxes filled with blue), while the coupling and cohesion values of the rest of the modules remain unchanged (presented in rectangular boxes filled with gray). Then, the FCB value is updated accordingly.

Similarly, the incremental evaluation of the cohesion and coupling values of a solution (objectives 1 and 2 in the MCA and ECA problems) can be performed by storing the coupling and cohesion values of each module. Regarding the size difference between the smallest and the largest modules (objective 5 in ECA), the efficient evaluation is again based on the storage of the size values of each module. After a move operation, the size of the affected modules is updated. Finally, the number of modules (objective 4 in MCA and

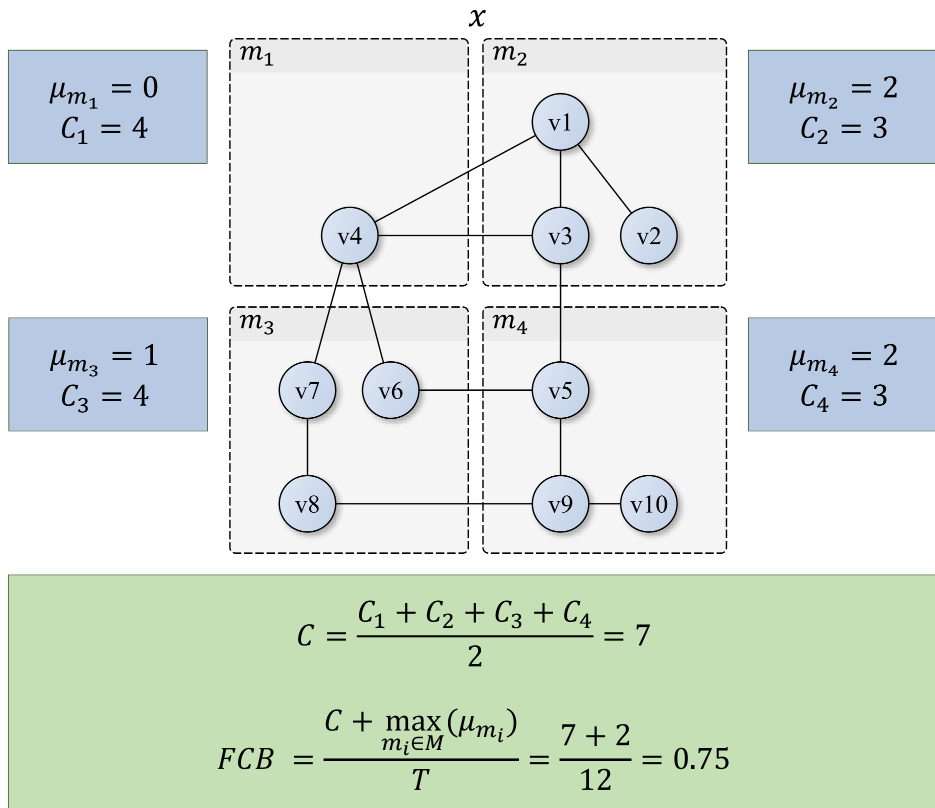


Figure 4.13 Calculation of the FCB value of a solution x.

ECA) and the number of isolated modules (objective 5 in MCA) are trivially calculated.

4.6.2 Identification of promising areas in the search space

Ideally, an approximate algorithm should be able to identify promising areas of the search space, so that the optimal solution is found by exploring only a subset of solutions. The narrower the search space explored, the faster the search process. Following this idea, we present here a strategy to identify promising areas in the search space with the aim of improving the efficiency of the search procedures proposed in previous sections.

This strategy is based on the following theorem, stated by Köhler et al.: "*Let $G(V, E)$ be an input MDG to the SMCP and let $A(v)$ be the set of adjacent vertices of $v \in V$. Suppose that in the optimal solution of the SMCP, all vertices in $A(v)$ are assigned to at*

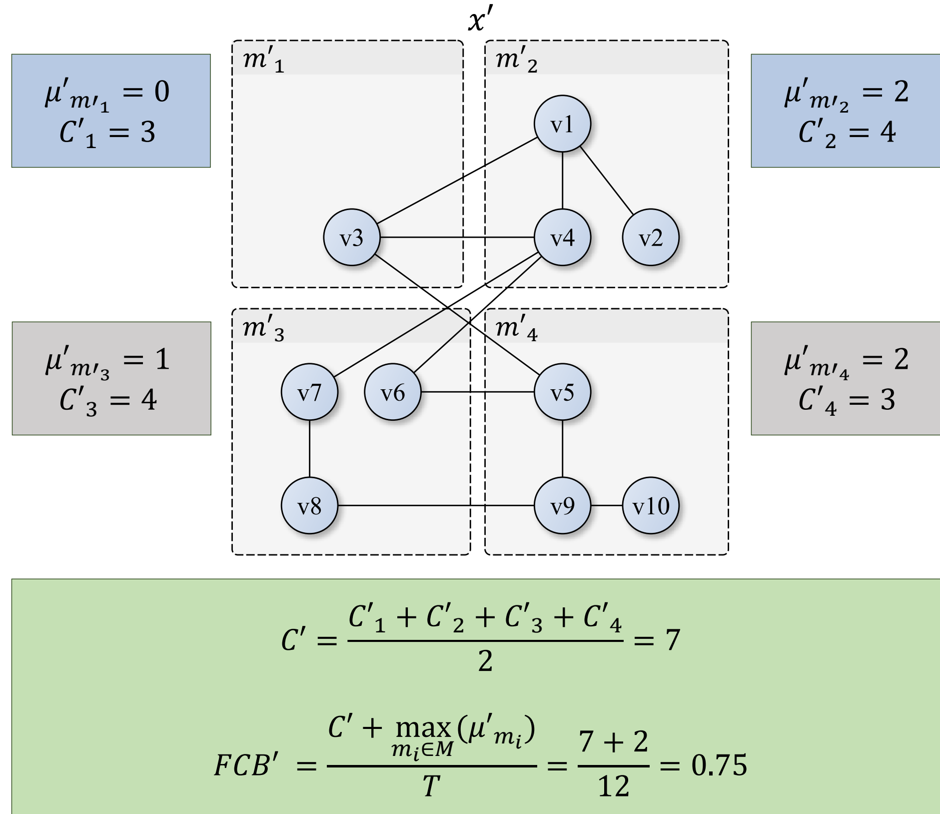


Figure 4.14 Incremental evaluation of the FCB value of a solution x', obtained after applying the operation $x' \leftarrow Swap(x, v_3, v_4)$.

most two different modules m_i and m_j. Then, v is either assigned to m_i or to m_j" [75]. Based on this theorem, we can reduce the size of the neighborhoods presented in Section 4.5 for the TurboMQ problem. In particular, we restrict the neighborhoods to consider only operations that relocate vertices in modules where there is at least one vertex adjacent to them. In Figure 4.15, we present an example to illustrate this behavior. In Figure 4.15(a), we represent the possible insert operations to relocate vertex $v4$ in the solution x. As can be observed, there are three possible insert operations, one for each module where vertex $v1$ can be relocated to (m_2, m_3, and m_4). On the right side, we calculate the change in the TurboMQ value that would be obtained when performing any of the move operations. As can be observed, only the first two insert operations (highlighted in green) would result in an improvement of the objective function. The third operation (highlighted in red)

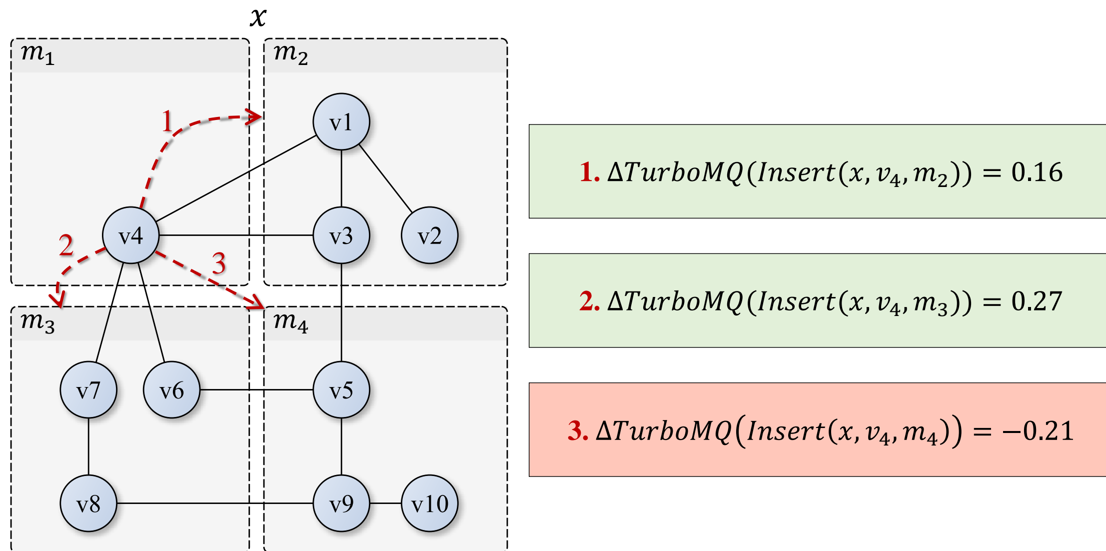
would result in a deterioration of the objective function. In Figure 4.15(b), we represent the promising insert operations to relocate vertex $v4$. That is, operations that relocate vertex $v4$ to a module where there is at least one vertex adjacent to $v4$. As can be observed, there exist only two promising insert operations under the premise mentioned above. Again, on the right side, we calculate the change in the TurboMQ value that would be obtained when performing any of the move operations. As can be observed, the operation that resulted in a deterioration of the objective function was regarded unpromising, since none of the vertices in m_4 was adjacent to $v4$. As a result, the move is not explored.

The aforementioned strategy, although designed for MQ, can be extended to the FCB, coupling, and cohesion metrics. In the case of the coupling and cohesion metrics, it is trivial to see that moving a vertex from a module where there are no adjacent vertices to a module where there is at least one adjacent vertex always results in an improvement. Indeed, being $\gamma(v, m)$ the sum of the weights of the edges that connect v to vertices that are located in module m, then any move that relocates vertex vi from module m_s to module m_t results in an improvement in both coupling and cohesion only if $\gamma(v, m_t) > \gamma(v, m_s)$.

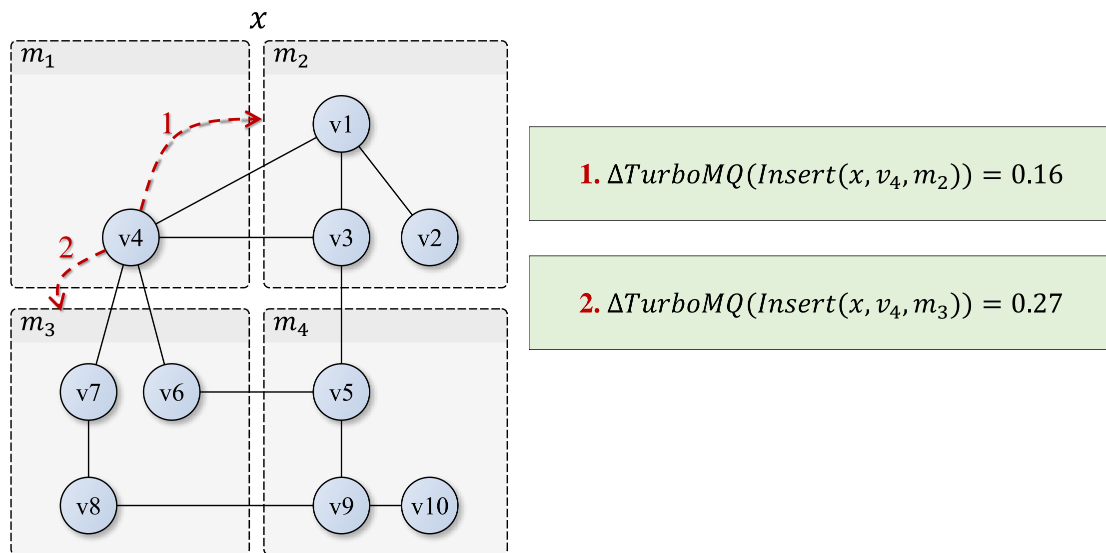
In the case of FCB, it is possible to find unpromising insert operations that do not deteriorate the FCB value of the solution (only if the vertex to be relocated belongs to the module with maximum cohesion). However, it is not possible to find a non-promising insert operation that results in an improvement of the FCB value. Therefore, by exploring only promising regions of the search space as described above, only non-improving move operations are discarded.

The described strategy allows the algorithms to explore only promising regions of the search space, reducing the size of some of the neighborhoods, and, therefore, enhancing the efficiency of the search process. In particular, the complexity of exploring the neighborhoods N_1, N_2, N_3, and N_5, proposed in Section 4.5, using this strategy, is reduced as follows:

1. The first neighborhood (N_1), described in Section 4.5.1, is based on insert operations. Since promising moves only insert a vertex in adjacent modules, the complexity is reduced from $\mathcal{O}(|V| \cdot |M|)$ to $\mathcal{O}(|V| \cdot a)$, where a is the average number of adjacent modules per vertex.



(a) Evaluation of the TurboMQ value of the possible insert operations involving vertex $v4$ in solution x.



(b) Evaluation of the TurboMQ value of the promising insert operations involving vertex $v4$ in solution x.

Figure 4.15 Identification of promising insert operations involving vertex $v4$ in solution x and evaluation of the TurboMQ value that results from the operations.

2. The second neighborhood (N_2), described in Section 4.5.1, is based on swap operations. Since promising moves only insert a vertex in adjacent modules, the complexity of exploring this neighborhood is reduced from $\mathcal{O}(|V|^2)$ to $\mathcal{O}(|V| \cdot a \cdot d)$, where a is the average number of adjacent modules per vertex and d is the average number of vertices per module. As the number of modules adjacent to any vertex cannot be greater than the total number of modules ($a \leq |M|$) and $|V| = |M| \cdot d$, then it is clear that $a \cdot d \leq |V|$. In practice, it is usually the case that $a \cdot d < |V|$, unless the graph is fully connected.

3. The third neighborhood (N_3), described in Section 4.5.2, is based on extraction operations. In this case, since a promising move operation will only place a vertex in a module together with adjacent vertices, the complexity of exploring this neighborhood is reduced from $\mathcal{O}(V \cdot (V-1) + V \cdot (V-1) \cdot (V-2))$ to $\mathcal{O}(|V| \cdot q + V \cdot q \cdot (q-1))$, where q is the average number of adjacent vertices per vertex. Of course, $q \leq |V|$. Again, it is hardly ever the case that $q = |V|$.

4. The fifth neighborhood (N_5), described in Section 4.5.3, is based on merge operations. In this case, the complexity of exploring this neighborhood is reduced from $\mathcal{O}(|M| \cdot \frac{|M-1|}{2})$ to $\mathcal{O}(|M| \cdot \frac{z}{2})$, where z is the average number of adjacent modules per module (two modules are considered adjacent if at least one vertex of one module is adjacent to one vertex of the other module). Of course, $z \leq |M|$. Again, it is hardly ever the case that $z = |M|$.

As can be noticed, the benefit of this strategy depends on the density of the graphs. The more dense the graph, the higher the number of adjacent vertices per vertex and the lower the number of non-promising moves discarded. On the contrary, the more sparse the graph, the greater the reduction in the size of the neighborhoods. Nevertheless, in the case of real software projects, this strategy is particularly useful, since dependency graphs are frequently sparse, with densities ranging from 1.77 % to 21.52 % [98, 141]. Particularly, in the dataset used in this work, the average number of vertices is 156.37, while the average number of adjacent vertices per vertex is 10.22. Therefore, at least for this set of instances, $q \ll |V|$.

The described strategy is suitable for identifying promising regions of the search space when considering TurboMQ, FCB, coupling, and cohesion metrics. However, it is not applicable when considering the number of modules, the number of isolated modules, or the size difference between the largest and smallest modules in a solution. Instead, for these objectives, we propose a reduction of the search space to be explored by leveraging the categorization of neighborhoods described in Section 4.5.

In the case of the number of modules, we only explore neighborhood structures where the move operations are designed to increment the number of modules in the solution. That is, we only explore neighborhoods N_3 and N_6, based on extraction and split operations.

In the case of the number of isolated modules, we only explore neighborhood structures where the move operations are designed to decrease the number of modules in the solution. That is, we only explore neighborhoods N_4 and N_5, based on destruction and merge operations. Moreover, we only explore move operations that destroy isolated modules.

Finally, in the case of the size difference between the largest and smallest modules in a solution, we avoid exploring the second neighborhood structure, N_2, since it is based on swap operations, which do not alter the size of the modules. For the rest of the neighborhood structures we only consider moves that affect modules with the largest or the smallest size in the solution.

4.6.3 Analysis of the contribution of the guiding functions

The use of multiple conflicting criteria to evaluate efficient points in MOPs benefits an accurate representation of the desirable properties of good efficient points. However, this comes at a cost. The larger the number of objectives considered, the larger the number of efficient points in the Pareto optimal solution [134]. In fact, the number of efficient points in a Pareto optimal solution may be exponential with respect to the size of the problem [41]. Furthermore, the number of objectives considered as guiding functions has an impact on the computing time of MO-VNS-based approaches. As can be observed in Algorithm 11, presented in Section 4.4, the number of iterations through the first loop of the algorithm (step 4) directly depends on the number of objectives considered ($|R|$). Indeed, the algorithm explores the set of neighborhood structures N for each efficient point $x \in SE$

considering each objective $i \in R$. The larger the set of objectives considered, the greater the computing time needed by the procedure.

Usually, the objective function of a problem is used as the guiding function for the search process. In that case, the criterion for improving efficient points is given by the value of the objective function. However, the objective functions and the guiding functions do not need to be the same. We refer to objective functions as the criteria used to evaluate the quality of an efficient point, whereas guiding functions are used within the search process to guide the search towards promising areas of the search space. For example, the Variable Formulation Search methodology uses alternative functions to guide the search when multiple solutions in the neighborhood have the same value of the objective function [110].

Taking into account the aforementioned problems, we investigate the use of the objective functions proposed in the MCA (see Section 2.3) and ECA (see Section 2.4) problems as guiding functions. In particular, the aim here is to reduce the number of guiding functions used during the search. Ideally, the objective is to reduce the computing time of the search process without reducing the quality of the solution found. However, this might be complicated if the objective functions are conflicting. Therefore, a trade-off might be needed.

In 2017, Yuan et al. proposed a methodology to reduce the number of objectives in MOPs [140]. In particular, they tackled the task as an optimization problem with two objectives: (i) minimize the number of objective functions and (ii) minimize the error rate. Given a set of non-dominated efficient points, they proposed three different measures to calculate the error rate of each subset of objectives: δ, η, and γ. This measures are based on the dominance structure of the Pareto front before and after reducing the set of objectives or on the correlation among objectives.

Although Yuan et al. proposed an evolutionary algorithm to find good subsets of objectives, we consider that the number of objectives (5) in both MCA and ECA is not very large. Therefore, we calculate the error rate measures for every possible subset of objectives for both MCA and ECA. Then, the most promising subsets of objectives are tested as guiding functions in preliminary experiments to study the trade-off between computing time and solution quality when using a reduced set of objectives as guiding functions. The

experiments are presented in Section 5.2.3. Note that the aim is to reduce the number of objectives used as guiding functions, not the set of objectives used as evaluation functions. At the end of the search process, the efficient points obtained are evaluated using the entire set of objectives proposed in MCA or ECA.

Chapter 5

Experiments

In the area of operations research, the performance of an algorithm must be empirically evaluated. In [14], an experiment is defined as "*a set of tests run under controlled conditions for a specific purpose: to demonstrate a known truth, to check the validity of a hypothesis, or to examine the performance of something new*". Experiments designed to evaluate an algorithm consist of solving a set of instances for a problem using an implementation of the algorithm studied. The goal of each experiment must be clearly stated prior to performing the experiment. Usually, experiments in this area are designed to either (i) analyze the behavior of an algorithm and configure its free components or (ii) compare its performance with other algorithms for a particular set of problems [14].

In this section, we present the experiments performed to analyze the performance of the algorithms presented in this doctoral thesis. These experiments are divided into two sections: first, in Section 5.2, we present some preliminary experiments designed to analyze and configure the proposed algorithms; then, in Section 5.3, the performance of the proposed approaches is compared with those from the best algorithms available in the literature.

5.1 Dataset

For the computational experiments, we have used a set of 124 real software instances proposed by previous works [104]. All the instances included in the dataset are real-world

software projects, extracted from their public repositories. These instances are of varying sizes. Once modeled as MDGs, these instances have between 2 and 1161 vertices and between 2 and 11722 edges. On average, these instances have 156.37 vertices (with a standard deviation of 216.72) and 948.79 edges (with a standard deviation of 1751.86). In the work where the dataset was proposed for the first time, the instances were divided into four different categories according to their size: 64 small instances (up to 68 vertices), 29 medium instances (from 74 to 182 vertices), 18 large instances (from 190 to 377 vertices), and 13 very large instances (from 413 to 1161 vertices). In Appendix A, we describe in detail the instances included in the dataset and their characteristics.

In Figure 5.1, we show the relationship between the number of vertices (on the horizontal axis) and the number of edges (on the vertical axis) in the instances of the dataset. Each instance is represented by a red dot. As can be observed, the number of edges increases linearly as the number of vertices grows. We represent this trend with a dashed line. As the dataset is composed of real software projects, this relation shows that software components usually have a few dependencies on other components. As more components are added to the projects, the number of dependencies grows linearly because each component depends only on a few other components in the project. That is, the number of dependencies per component remains constant as the graphs grow in size. If each component that was added depended on all already existing components of the project, the number of edges would grow exponentially. In Figure 5.2, we show the relationship between the number of vertices (on the horizontal axis) and the density (on the vertical axis) of the instances in the dataset. Again, each instance is represented by a red dot. As can be observed, due to the number of edges per vertex being constant, the density of the graphs tends to be reduced as the graphs grow in size. Although there are some exceptions, the dependency diagrams of real software projects tend to be sparse, as shown in the aforementioned figures, a fact that has already been pointed out in the related literature [98]. In this case, the instances in the dataset have an average density of 8.83%, with a standard deviation of 11.65%. These characteristics are of particular interest for the design of the advanced strategy presented in Section 4.6.2 to reduce the size of neighborhood structures. Since that strategy relies on the adjacency of vertices, the sparser the instances, the greater the reduction in the size of the neighborhood structures.

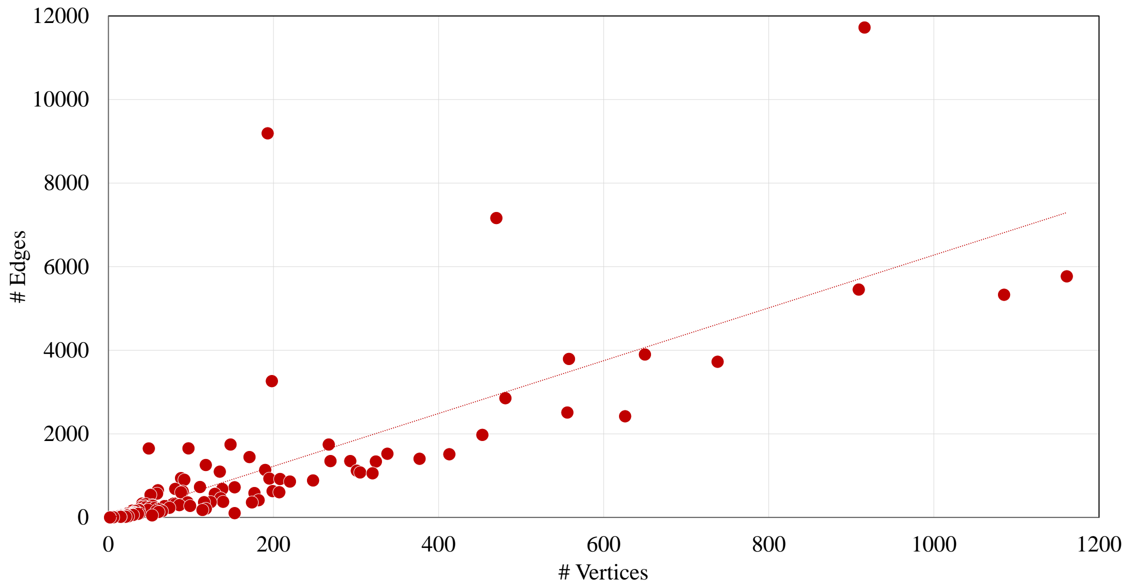


Figure 5.1 Number of vertices and edges of each instance in the dataset.

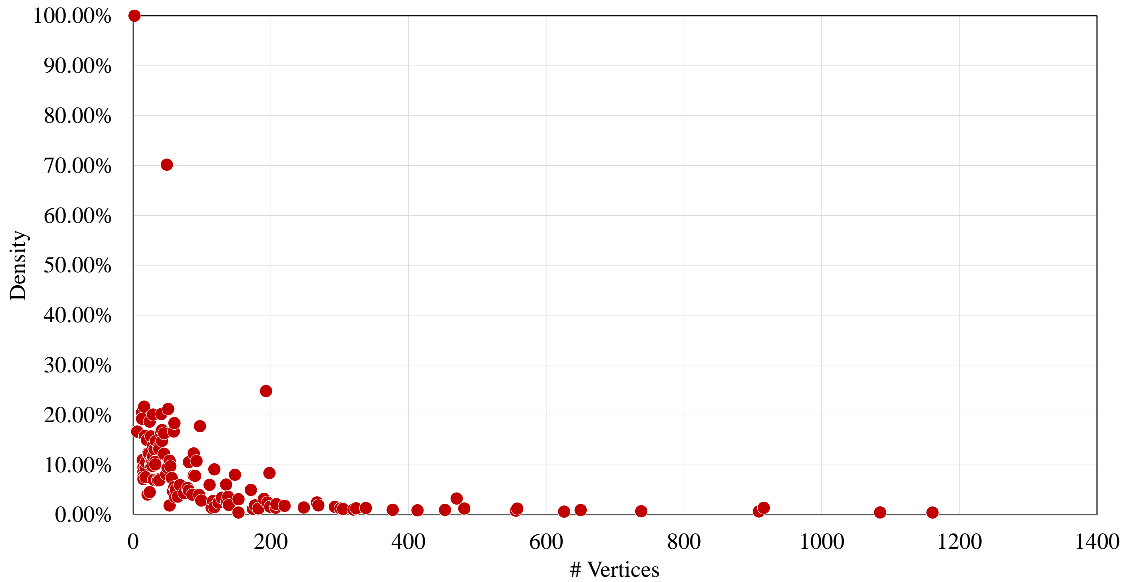


Figure 5.2 Number of vertices and density of each instance in the dataset.

5.2 Preliminary computational experiments

In this section, we present some preliminary experiments designed to configure and analyze the proposed algorithmic approaches. In particular, experiments designed to analyze the GRASP-VND approach are presented in Section 5.2.1, experiments designed to analyze the GVNS approach are presented in Section 5.2.2, and experiments designed to analyze the MO-GVNS approach are presented in Section 5.2.3.

5.2.1 GRASP-VND

In this section, we present the preliminary computational experiments performed to analyze the GRASP-VND approach, which focuses on the TurboMQ problem. All the experiments presented in this section have been executed on the same computer environment, a Microsoft Windows 10 Pro 10.0.19042 x64 with an AMD EPYC 7282 @ 2795 MHz CPU with 8 cores and 8 GB RAM. The algorithm was implemented with Java OpenJDK 15.0.2. For the preliminary experiments, 14 instances have been selected at random from the whole dataset, following a similar distribution in terms of size. That is, for the reduced dataset, 8 instances have been selected from the small group, 2 instances from the medium group, 2 instances from the large group, and 2 instances from the very large category. In particular, the instances included in the reduced dataset for the preliminary experiments are the following: *apache_ant_taskdef, cia, gae_plugin_core, joe, jscatterplot, jtreeview, jxlsreader, lwjgl-2.8.4, mod_ssl, net-tools, nmh, regexp, star* and *wu-ftpd-1*. The smallest instance in the reduced dataset, *regexp*, has 14 vertices and 20 edges. The largest instance, *apache_ant_taskdef*, contains 626 vertices and 2421 edges. The average number of vertices is 157.79, with an average deviation of 184.86. The average number of edges is 838.43, with an average deviation of 1019.11. The density of the instances is between 0.62% and 21.18%, with an average density of 7.40% and a standard deviation of 5.68%.

Neighborhood structure	$\triangle$ Avg. F.O.	Avg. CPUt (s)
N_1	0.6258	<0.01
N_2	0.0207	<0.01
N_3	0.4663	0.04
N_4	0.1133	0.01

Table 5.1 Contribution of exploring each neighborhood to the quality of the initial solutions obtained by the constructive procedure.

Contribution of each neighborhood structure

The first preliminary experiment is designed to evaluate the contribution of exploring each neighborhood structure to the performance of the algorithm. To do so, we analyze the results obtained by exploring each neighborhood structure within the GRASP-VND approach in isolation. In particular, we design four different variants of the GRASP-VND method, each exploring only one neighborhood structure (N_1, N_2, N_3, or N_4). Each method is run for a hundred iterations per instance. In the end, we report the average quality difference between the initial solution built by the constructive procedure and the best solution found by the VND component. Importantly, we ensure that the initial solution built by the constructive procedure at each iteration is the same for all four variants, in order to perform a fair comparison. In Table 5.1, we report the results obtained. In particular, for each neighborhood structure, we report the average improvement achieved by the VND component ($\triangle$ Avg. F.O.) and the average computing time consumed (Avg. CPUt (s)). As can be observed, the exploration of the neighborhood structures N_1 and N_3 yielded the best results, with an average improvement of 0.6258 and 0.4663, respectively. Exploring the fourth neighborhood structure, N_4, resulted in a modest improvement of 0.1133. On the contrary, exploring the neighborhood structure N_2 resulted in a low improvement, with an average of 0.0207. Taking into account the results obtained, we decided to configure the GRASP-VND algorithm to explore only neighborhood structures N_1, N_3, and N_4.

Order of the neighborhoods

After analyzing the contribution of exploring each neighborhood structure to the quality of the solutions in the previous section, we configured the algorithm to explore only N_1, N_3,

Order	Avg. F.O.	Avg. Dev. (%)	# Best	Avg. CPUt (s)
N_1, N_3, N_4	15.9545	<0.01%	12	6.23
N_1, N_4, N_3	15.9549	<0.01%	11	5.78
N_3, N_1, N_4	15.9537	0.02%	9	10.36
N_3, N_4, N_1	15.9541	<0.01%	10	15.37
N_4, N_1, N_3	15.9542	<0.01%	9	9.33
N_4, N_3, N_1	15.9641	<0.01%	10	15.41

Table 5.2 Comparison of the results obtained with the GRASP-VND procedure with different orderings of the proposed neighborhoods in the reduced dataset.

and N_4. As mentioned previously, VND performs a systematic search within a set of neighborhood structures. After successfully improving the best solution found, VND restarts the search from the first neighborhood structure in the set, and this process is repeated until all neighborhood structures have been explored without finding an improvement. Therefore, the order in which the neighborhood structures are explored within the VND component can have a considerable impact on the performance of the algorithm.

In this experiment, we investigate which ordering of the considered neighborhoods is the best one for exploring the neighborhood structures. We configure six different variants of the GRASP-VND algorithm, each exploring the neighborhood structures in a different order. Then, each variant is run for a hundred iterations per instance. Again, to perform a fair comparison, we ensure that every variant starts from the same set of initial solutions for each instance.

In Table 5.2, we report the results obtained. For each variant, we report the average quality of the best solutions found (Avg. F.O.), the average deviation of the best solution found from the best solution found by any variant (Avg. Dev. (%)), the number of best solutions found (# Best), and the average computing time consumed (Avg. CPUt (s)). As can be seen, the differences in terms of quality are low. On the contrary, some orderings resulted in a considerable increase in computing time. Due to the number of best solutions found and the short computing time needed, we decided to configure the GRASP-VND algorithm to explore the neighborhood structures in the following order: N_1, N_3, and N_4.

Stopping criterion

In optimization, a trade-off must be made between computing time and solution quality [14]. In this experiment, we analyze the behavior of the algorithm in this regard and, accordingly, set a stopping criterion. Since the algorithm is a multistart procedure, a maximum number of iterations seems like a natural stopping criterion, although others (e.g., maximum computing time) could have been considered as well. Here, we execute the algorithm with the configuration set in previous experiments. That is, GRASP-VND explores neighborhood structures N_1, N_3, and N_4, in this order. For the experiment, a maximum number of 100 iterations per instance is set as the stopping criterion for the algorithm. At the end of each iteration, the quality of the best solution found during the search process is reported.

In Figure 5.3, we represent the results obtained in the aforementioned experiment. In particular, we represent the average deviation (Avg. Dev. (%)) from the best solution found in each iteration for each instance to the best solution found for each instance during the entire search. We represent the average deviation for all instances in the preliminary dataset with a solid black line. In addition, we represent the average deviation at each iteration for the groups of small (blue dashed line), medium (green dash-dotted line), large (yellow long dash-dot-dotted line), and very large (orange long dash line) instances separately. As can be observed, the best solutions for the small and medium instances are already found in the fifteenth iteration. On the contrary, the best solutions for the largest instances in the reduced dataset are found in iteration 81 approximately. However, the improvement obtained after ten iterations is quite small ($<$0.05%). The results obtained show that the proposed method quickly converges to good solutions after ten iterations. Due to these results, in order to find a balance between computing time and solution quality, we decide to set a stopping criterion of twenty iterations per instance for the proposed approach.

Influence of the advanced strategies

Here, we conduct an experiment to analyze the impact of implementing the advanced strategies described in Section 4.6 into the GRASP-VND procedure. In particular, we test the influence of the incremental evaluation of TurboMQ (Section 4.6.1) and the reduction of

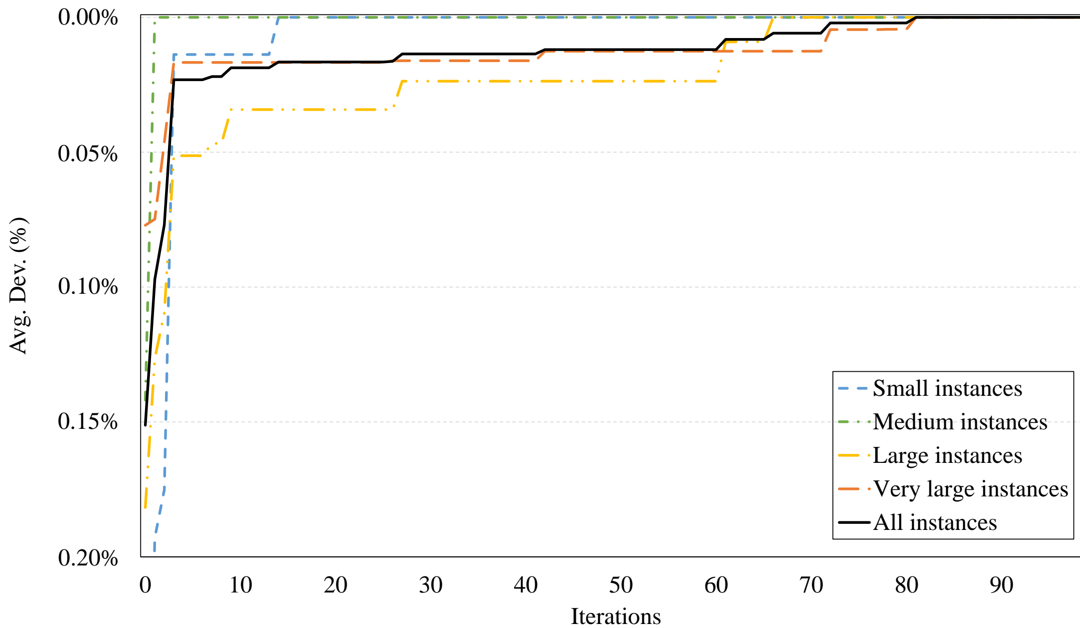


Figure 5.3 Average deviation (%) of the best solution found by the algorithm at each iteration to the best solution found after 100 iterations for each the instances of the preliminary dataset.

the size of the neighborhood structures (Section 4.6.2). To analyze their impact, we execute the algorithmic proposal with three different configurations: (i) without advanced strategies in place; (ii) implementing the incremental evaluation of TurboMQ; and (iii) implementing both the incremental evaluation and the reduction of the size of neighborhood structures. For every variant of the algorithm, the neighborhood structures N_1, N_3, and N_4 are explored in that order, and the stopping criterion is set to a maximum of twenty iterations. Of course, to ensure a fair comparison, every variant starts from the same set of initial solutions for each instance.

In Table 5.3, we present the results obtained. For each variant, we report the average quality of the best solutions found (Avg. MQ), the average deviation from the best solution found by each configuration to the best solution found by any of the configurations (Avg. Dev.), the number of best solutions found (# Best), and the average computational time (Avg. CPUt (s)). As can be observed, all variants achieved solutions of similar quality.

Advanced strategies	Avg. MQ	Avg. Dev.	# Best	Avg. CPUt (s)
(i) None	15.9528	0.01%	12	147,632.44
(ii) IE	15.9528	0.01%	12	182.31
(iii) IE+RNS	15.9519	<0.01%	11	1.71

Table 5.3 Performance of the GRASP-VND procedure when implementing none of the advanced strategies, denoted as "(i) None", only the incremental evaluation, denoted as "(ii) IE", and both the incremental evaluation and the reduction of the size of the neighborhood structures, "(iii) IE+RNS".

However, the differences in terms of computational time are considerable. By implementing the incremental evaluation of the objective function, the computational time is reduced by three orders of magnitude, from 147,632.44 seconds on average to just 182.31 seconds. That is, a reduction of 99.88% of the computational time is achieved. When both the incremental evaluation and the reduction of the size of neighborhood structures are implemented, the reduction of computational time is further improved, from 182.31 seconds to just 1.71 seconds on average. That is, an additional reduction of 99.06% is achieved when the second advanced strategy is also implemented.

Notice that there exists a slight difference in the quality of the solutions found by the second and third configurations. This difference can be explained by the reduction in the size of the neighborhood structures considered. Since a first improvement strategy is implemented in the local search procedures within the VND component, the search patterns differ when the neighborhood structures are different.

5.2.2 GVNS

In this section, we present the preliminary experiments devoted to analyze and configure the GVNS method proposed in Section 4.3 for the FCB problem. To perform fair comparisons, all experiments have been executed in the same computing environment: a Microsoft Windows 10 Pro 10.0.19042 x64 operating system, with an AMD EPYC 7282 @ 2795 MHz CPU with 8 cores and 8 GB RAM. Additionally, the proposed method has been implemented in Java 17.0.1, using the Metaheuristic Optimization framewoRK (MORK) project [90].

For the preliminary experiments, 14 instances have been selected at random from the whole dataset, following a similar distribution in terms of size. That is, for the reduced dataset, 7 instances have been selected from the small group, 3 instances from the medium group, 2 instances from the large group, and 2 instances from the very large category. In particular, the instances included in the reduced dataset for the preliminary experiments are the following: *apache_ant_taskdef, gae_plugin_core, javacc, joe, jscatterplot, jtreeview, jxlsreader, lwjgl-2.8.4, mod_ssl, net-tools, nmh, regexp, star* and *wu-ftpd-1*. The smallest instance in the reduced dataset, regexp, has 14 vertices and 20 edges. The largest instance, *apache_ant_taskdef*, contains 626 vertices and 2421 edges. The average number of vertices is 166.00, with an average deviation of 181.65. The average number of edges is 876.79, with an average deviation of 1002.60. The density of the instances is between 0.62% and 21.18%, with an average density of 6.68% and a standard deviation of 5.53%.

Contribution of each neighborhood structure

This first experiment is devoted to analyze the contribution of exploring each neighborhood structure to the overall performance of the algorithm. To do so, we analyze the results obtained by exploring each of the neighborhood structures in isolation within the GVNS approach. In particular, we design four different variants of the GVNS procedure, each exploring only one neighborhood structure (N_1, N_2, N_3, N_4, N_5, or N_6). Each method is run for a hundred iterations per instance. Importantly, we ensure that the initial solution at each iteration is the same for all four variants, in order to perform a fair comparison.

In Table 5.4, we report the results obtained. In particular, for each neighborhood structure, we report the category of the neighborhood (Category), as described in Section 4.5 (category 1 denotes neighborhoods that maintain the number of modules, category 2 denotes neighborhoods that increase the number of modules, and category 3 denotes neighborhoods that reduce the number of modules); the average quality of the solutions found (Avg. O.F.); and the average computing time consumed (Avg. CPUt (s)). In addition, we include the average quality of the initial solutions for comparison purposes. As can be observed, the exploration of the neighborhood structures N_1 and N_4 obtained the best results, with an average solution quality of 0.6753 and 0.6824, respectively (let us remind the reader that the objective is to minimize the value of the FCB objective function). Exploring the rest

Neighborhood structure	Category	Avg. O.F.	Avg. CPUt (s)
None	-	0.9524	0.04
N_1	1	0.6753	3.33
N_2	1	0.8425	0.36
N_3	2	0.8862	0.94
N_6	2	0.9494	0.06
N_4	3	0.6824	0.11
N_5	3	0.8059	0.06

Table 5.4 Contribution of exploring different neighborhoods in isolation within the GVNS procedure to the search process.

of the neighborhood structures resulted in a modest average quality. Regarding the computational time, the exploration of the first neighborhood structure (N_1) is the most costly, with an average computing time of 3.33 seconds. Given the results obtained, we decided to configure the algorithm to explore neighborhood structures N_1 and N_4. Moreover, since the main idea of VNS is to explore multiple neighborhood structures, we decided to complement the neighborhoods by configuring the algorithm to also explore N_3, which is the best neighborhood structure from the second category. Therefore, the algorithm is configured to explore a neighborhood structure for each identified category, giving the algorithm flexibility to reduce or increase the number of modules.

Order of the neighborhoods

In this section, we investigate which is the best ordering to explore the neighborhood structures. We study all possible orderings for the exploration of the neighborhood structures. We configure six different variants of the GVNS algorithm, each exploring the neighborhood structures in a different order. Again, to perform a fair comparison, we ensure that every variant starts from the same initial solution for each instance. In Table 5.5, we report the results obtained. For each possible order, we report the average quality of the best solutions found by each variant (Avg. O.F.), the average deviation of the best solution found by each variant from the best solution found by any variant (Avg. Dev. (%)), the number of best solutions found (# Best), and the average computing time consumed (Avg. CPUt (s)). As can be seen, although the differences in terms of quality are low, the last order (N_4, N_3,

Order	Avg. O.F.	Avg. Dev. (%)	# Best	Avg. CPUt (s)
N_1, N_3, N_4	0.6055	6.93 %	3	1.94
N_1, N_4, N_3	0.6053	6.89 %	3	2.85
N_3, N_1, N_4	0.6163	8.40 %	3	2.29
N_3, N_4, N_1	0.6045	5.96 %	4	0.89
N_4, N_1, N_3	0.6093	7.53 %	4	**0.15**
N_4, N_3, N_1	**0.5971**	**4.88 %**	**7**	0.18

Table 5.5 Comparison of the results obtained with the GVNS algorithm with different orderings of the proposed neighborhoods in the reduced dataset.

and N_1) results in solutions of better quality, achieving a deviation of almost 4% less than the worst ordering. Moreover, the time consumed is almost identical to that of the fastest order tested. Due to these results, we decided to configure the GVNS algorithm to explore the neighborhood structures in the following order: N_4, N_3, and N_1. The results obtained indicate that the ordering of neighborhood structures when multiple neighborhoods are used is something worth exploring.

Comparison of shake procedures

The shake procedure within a GVNS method is devoted to introduce some diversification into the search process. Within the shake procedure, a neighborhood structure is explored stochastically. In this experiment, we analyze the performance of the GVNS method by exploring different neighborhood structures in the shake procedure. In particular, we configure three different variants of the algorithm. All variants explore the same set of neighborhood structures within the VND component in the following order: N_4, N_3, and N_1. However, they differ in the neighborhood structure explored within the shake procedure: the first variant explores the neighborhood structure N_2, the second explores N_6, and the third explores N_5.

Since the neighborhoods considered for the shake procedures are considerably different (an operation in N_2 affects two vertices, an operation in N_5 affects all the vertices of two modules, and an operation in N_6 affects half of the vertices of a module), the magnitude of the perturbation performed in the solution, given the same value of k, might also

Table 5.6 Comparison of different neighborhoods explored within the shake component of the GVNS method. Each variant compared explores one of three neighborhood structures (N_2, N_5, or N_6) within the shake procedure and uses one of three values (10, 20, or 30) for the parameter k_{max}.

	Avg. O.F.			Avg. Dev. (%)			# Best			Avg. CPUt (s)		
k_{max}	10	20	30	10	20	30	10	20	30	10	20	30
$Shake_1$	0.57	**0.56**	**0.56**	**0.37**	**0.15**	**0.15**	**12**	**13**	**13**	3.79	4.86	6.31
$Shake_2$	0.59	0.59	0.59	5.31	5.63	5.63	3	2	2	1.92	1.72	2.04
$Shake_3$	0.57	0.57	0.57	1.41	1.73	1.73	5	3	3	8.29	18.56	31.46

be different. Therefore, for each variant, we test three different values for the parameter k_{max}: 10, 20, and 30. Moreover, we adapt the size of the perturbation made by the shake procedure to the size of the instance at hand. In particular, at each step of the algorithm, the number of `Swap` moves performed by $Shake_1$ is $\max(k, (|V| * k)/100)$. The number of `Split` moves performed by $Shake_2$ is $\min(\max(k, (originalNumberOfModules * k)/100), originalNumberOfModules)$. Finally, the number of `Merge` moves performed by $Shake_3$ is $\min(\max(k, (originalNumberOfModules * k)/100), originalNumberOfModules)$.

In Table 5.6, we present the results obtained in the comparison of different shake methods. For each variant tested and each value of k_{max}, we present the average quality of the best solution obtained for each instance (Avg. O.F.), the average deviation from the best solution found in the experiment (Avg. Dev. (%)), the number of instances for which the best solution was obtained (# Best), and the average execution time consumed (Avg. CPUt (s)). Note that the comparisons between the shake procedures were made independently for each maximum value of k_{max} (i.e., the deviation and the best solutions were calculated separately for $k_{max} = 10$, $k_{max} = 20$, and $k_{max} = 30$). It is worth mentioning that every variant started from the same initial solution for each instance. As can be seen in the table, the first variant, which explored N_2 in the shake procedure, obtained the best results, outperforming the other variants in terms of quality for each value of k_{max} tested. Therefore, we configured the shake procedure within the GVNS method to explore the neighborhood structure N_2.

Maximum value of k

Here, we perform an experiment devoted to analyze the performance of the GVNS method with different values of the parameter k_{max}. Again, as in the previous experiment, the GVNS method is configured to explore the neighborhoods N_4, N_3, and N_1 within the VND component, in this order. Moreover, the neighborhood structure N_2 is explored within the shake procedure. Then, the GVNS is configured to run for a maximum of 300 seconds for each instance.

In Figure 5.4, we represent the average quality of the best solutions found by each variant of the method at any time during the execution. As can be observed, six different variants have been compared with the following values of k_{max}: 10, 20, 30, 40, 50, and 60. Depending on the time horizon considered, different values of k_{max} resulted in the best performance. For instance, the method configured with $k_{max} = 10$ achieved the best results in the first 50 seconds, while the method configured with $k_{max} = 20$ was the best configuration in the time interval between 50 and 100 seconds. Finally, the method configured with $k_{max} = 30$ was the best after 120 seconds. In this sense, the configuration of the algorithm should be set according to the particular running context. Although $k_{max} = 10$ allowed the algorithm to improve solutions faster than other values, $k_{max} = 30$ achieved better long-term performance. Therefore, we decided to set the value of k_{max} at 30 for the configuration of the approach.

Stopping criterion

To find a balance between the quality of the solutions and the computational effort, it is necessary to establish a stopping criterion in search algorithms. In this experiment, we analyze different stopping criteria for the proposed GVNS method. We propose a stopping criterion based on the number of iterations without improving the best solution found during the search process. That is, the method is stopped after it gets stuck at a local optimum. In particular, we compare four different values for the maximum number of iterations without improvement: 5, 10, 15, and 20. The rest of the algorithm is configured with the parameters selected in previous experiments: the shake procedure explores N_2, k_{max} is set to 30, and the VND approach explores the neighborhoods N_4, N_3, and N_1, in that order.

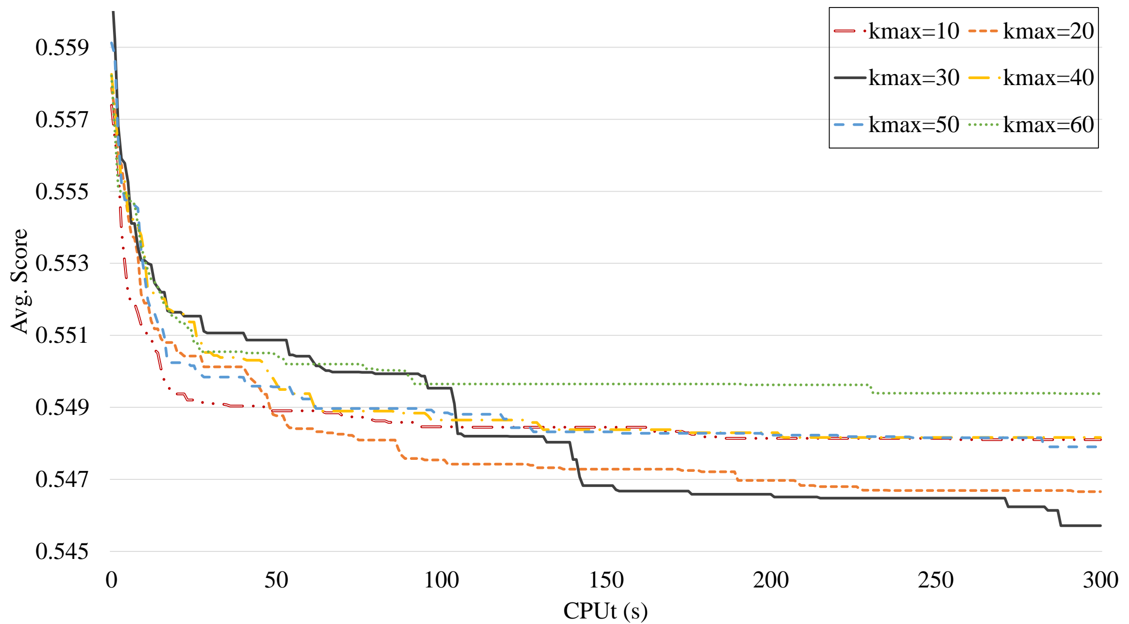


Figure 5.4 Comparison of the average quality of the best solution found for the instances in the reduced dataset over time with different values of k_{max}.

In Table 5.7, we present the results obtained for each stopping criterion (Iterations without improvement). As expected, a higher number of maximum iterations without improvement allowed the method to spend more time exploring the search space. However, the relative improvement obtained between consecutive configurations decreased as the number of maximum iterations increased. In fact, we barely observe an improvement in terms of quality by increasing the number of maximum iterations without improving from 10 to 15, and the improvement is non-existent after increasing this number from 15 to 20. Therefore, we decided to configure the stopping criterion of the proposed algorithm to stop the search after 15 consecutive iterations without improving the best solution found.

Influence of the advanced strategies

In this experiment, we test the impact of the advanced strategies, presented in Section 4.6, to the performance of the algorithm. For this experiment, we compare four variants of the GVNS approach. The first variant does not implement any of the advanced strategies. The second variant uses the efficient evaluation of the objective function, presented in Section

Table 5.7 Comparison of the results obtained by the GVNS method when configuring the algorithm to stop after 5, 10, 15, or 20 iterations without improvement.

Iterations without improvement	Avg. FCB	Avg. dev. (%)	# Best	Avg. CPUt (s)
5	0.5509	0.27 %	10	30.42
10	0.5501	0.09 %	12	54.57
15	0.5496	0.00 %	14	87.07
20	0.5496	0.00 %	14	99.95

4.6.1. The third variant uses the reduction of the size of the neighborhood structures, presented in Section 4.6.2. Finally, the fourth variant uses both the efficient evaluation of the objective function and the reduction of the size of the neighborhood structures. Again, the rest of the parameters of the algorithm are configured as reported in previous experiments. That is: the VND approach explores the neighborhoods N_4, N_3, and N_1, in that order; the shake procedure explores N_2; k_{max} is set to 30; and the stopping criterion is set to 15 iterations without improvement.

In Table 5.8, we present the results obtained for the comparison of the influence of different advanced strategies. For each variant, we report the average quality of the obtained solutions (Avg. O.F.) and the average computing time used by the algorithm (Avg. CPUt (s)). As can be seen, the use of any of the advanced strategies results in a considerable reduction in computational effort of two orders of magnitude. In particular, the use of an efficient evaluation (EE) reduced the time consumption of the algorithm from an average of 89479.66 seconds to just 509.63 seconds, while the reduction of the size of the neighborhoods (RN) reduced the time consumption from an average of 89479.66 seconds to just 250.59 seconds. When combining both strategies, the time consumption of the algorithm is further reduced by two orders of magnitude to an average of 6.56 seconds. That is, four orders of magnitude less than the time needed by the algorithm without any of the proposed strategies. In light of the results obtained, we configure the GVNS method to use both advanced strategies.

5.2.3 MO-GVNS

In this section, we perform some preliminary experiments devoted either to configure the parameters or analyze the performance of the MO-GVNS method proposed in Section 4.4

Table 5.8 Comparison of the performance of the GVNS method with different advanced strategies: None, efficient evaluation (EE), and reduction of the size of neighborhoods (RN).

Advanced strategies	Avg. O.F.	Avg. CPUt (s)
None	0.5762	89479.66
EE	0.5762	509.63
RN	0.5703	250.59
EE+RN	0.5703	6.56

for the MCA and ECA problems. To perform the preliminary experiments, we use a subset of the instances available in the dataset described in Section 5.1. In particular, ten instances have been chosen from the aforementioned dataset: *bison, bunch2, cia, crond, dot, forms, jscatterplot, mailx, micq,* and *netkit-ftp*. The average number of vertices is 45.60, with a standard deviation of 16.87. The average number of dependencies is 196.60, with a standard deviation of 74.36. The average density of the resulting MDGs of the instances is 11.57 $\%$, with a standard deviation of 5.31 $\%$.

To analyze the quality of the sets of efficient points found by the algorithm, we use four quality indicators commonly used in MOPs: Hypervolume (HV), Coverage (C), generalized spread (Spread), and Inverted Generational Distance plus (IGD+). Given a reference front (R) and a set of efficient points (SE), multi-objective quality indicators usually measure either the convergence (i.e., how close is SE to R) or the diversity (i.e., how well distributed are the efficient points across the entire front) of the obtained set of efficient points. The HV indicator, which is one of the most widely used indicators in the literature, evaluates both the convergence and diversity. The higher its value, the better the front. In contrast, the C and IGD+ indicators measure the convergence of the solution obtained to a given reference front. In particular, C measures the percentage of efficient points in SE that are dominated by the reference set R, while IGD+ measures the proximity of the efficient points in SE to the efficient points in R. Finally, the Spread indicator measures the diversity of the efficient points contained in SE. For C, IGD+, and Spread, the lower their value, the better the quality of the front.

Note that for every quality indicator described above, a reference set is needed to evaluate a given solution. Ideally, the reference set would be the optimal Pareto front, but the

optimal Pareto front is frequently unknown. In the following experiments, we use an approximate reference front, which is obtained by combining the Pareto fronts generated by all the methods under comparison for each instance.

Comparison of shake procedures

In Section 4.4.2, we proposed four different shake procedures for the MO-GVNS approach. Here, we perform an experiment devoted to compare the performance of the algorithm when using each of the shake methods proposed. To perform the comparison, we configure the algorithm as follows: the MO-VND component explores neighborhoods N_1, N_3, N_2, and N_4, in that order; the stopping criterion is set to $k_{max} = 5$; in order to compare the convergence of the proposed method over time with the different shake procedures, we do not set a time limit in this experiment; finally, the objectives are tackled in the following order within the MO-VND component: MQ, cohesion, coupling, number of modules and number of isolated modules (in the case of MCA) / difference between the maximum and minimum size of modules (in the case of ECA). Then, four variants of the MO-GVNS method are compared, each one using a different shake procedure.

In Tables 5.9 and 5.10, we present the results obtained for both MCA and ECA, respectively. For each shake procedure, we report the average computing time (CPUt (s)), the average number of efficient points in the Pareto fronts obtained (PF size), the average hypervolume (HV), the average coverage (C), the average IGD+ (IGD+), and the average generalized spread (Spread). As can be seen, the first shake procedure obtains solutions that have better convergence to the approximate front, outperforming the other variants in three of the four quality indicators compared. Although it consumes more time than the other variants, this is due to the fact that the algorithm is able to explore a wider area of the search space before reaching the stopping criterion. Therefore, we configure the MO-GVNS method to use the first shake procedure, Shake 1, in the rest of the experiments.

Maximum value of k

In this section, we perform an experiment devoted to compare the performance of the algorithm with different values of the parameter k_{max}. As a reminder, this parameter specifies

Table 5.9 Comparison of different shake procedures for the MCA problem.

Shake	CPUt (s)	PF size	HV	C	IGD+	Spread
Shake 1	21428.59	748.10	**0.2903**	**0.1475**	**0.0024**	0.5665
Shake 2	266.16	373.00	0.2687	0.6389	0.0202	**0.5428**
Shake 3	8722.78	671.00	0.2868	0.3585	0.0053	0.5755
Shake 4	15307.98	688.30	0.2888	0.2589	0.0034	0.5578

Table 5.10 Comparison of different shake procedures for the ECA problem.

Shake	CPUt (s)	PF size	HV	C	IGD+	Spread
Shake 1	131874.37	1801.60	**0.2667**	**0.1821**	**0.0024**	0.5626
Shake 2	3626.04	989.30	0.2449	0.5905	0.0148	**0.5313**
Shake 3	39516.04	1587.90	0.2633	0.3293	0.0043	0.5583
Shake 4	101012.11	1758.50	0.2665	0.2181	0.0025	0.5546

the maximum value that the variable k can have during the search process. That is, the magnitude of the perturbation to be made within the shake procedure. In particular, we compare five different values for k_{max}: 1, 2, 3, 4, and 5. The rest of the configuration is the same for each variant: the first shake procedure, Shake 1, is used; the MO-VND component explores the neighborhoods N_1, N_3, N_2, and N_4, in this order; the stopping criterion is set to $k_{max} = 5$; in order to compare the convergence of the proposed method over time with the different shake procedures, we do not set a time limit in this experiment; finally, the objectives are tackled in the following order within the MO-VND component: MQ, cohesion, coupling, number of modules, and number of isolated modules (in the case of MCA) / difference between the maximum and minimum size of modules (in the case of ECA). Then, four variants of the MO-GVNS method are compared, each using a different shake procedure.

In Tables 5.11 and 5.12, we report the results obtained for MCA and ECA. For each variant, we report the average computing time (CPUt (s)), the average number of efficient points in the Pareto fronts obtained (PF size), the average hypervolume (HV), the average coverage (C), the average IGD+ (IGD+), and the average generalized spread (Spread). In addition to the aforementioned variants, we also introduce in the comparison the MO-VND method used within the MO-GVNS to test the contribution of using a MO-GVNS schema

Table 5.11 Comparison of different values of k_{max} for the MCA problem.

Method	k_{max}	CPUt (s)	PF size	HV	C	IGD+	Spread
MO-VND	NA	10.94	291.09	0.2586	0.4092	0.0217	**0.4835**
MO-GVNS	1	5072.68	624.70	0.2853	0.2925	0.0034	0.5589
MO-GVNS	2	7919.36	664.30	0.2873	0.2049	0.0017	0.5605
MO-GVNS	3	12790.68	705.60	0.2888	0.1163	0.0006	0.5650
MO-GVNS	4	19008.96	737.80	0.2898	0.0336	0.0001	0.5657
MO-GVNS	5	21428.59	748.10	**0.2903**	**0.0004**	$<$**0.0000**	0.5665

Table 5.12 Comparison of different values of k_{max} for the ECA problem.

Method	k_{max}	CPUt (s)	PF size	HV	C	IGD+	Spread
MO-VND	NA	26.42	435.90	0.2293	0.7394	0.0328	**0.4566**
MO-GVNS	1	22887.41	1528.40	0.2629	0.2682	0.0026	0.5534
MO-GVNS	2	42556.78	1641.70	0.2644	0.1675	0.0012	0.5590
MO-GVNS	3	69516.32	1702.60	0.2653	0.1040	0.0006	0.5584
MO-GVNS	4	111005.34	1773.80	0.2663	0.0316	0.0001	0.5597
MO-GVNS	5	131874.37	1801.60	**0.2667**	**0.0025**	$<$**0.0000**	0.5626

instead of just a MO-VND one. As can be observed, the combination of deterministic and stochastic exploration performed by any of the MO-GVNS configurations improves the results obtained by the MO-VND, which only performs a deterministic exploration. Then, the results obtained by the different MO-GVNS variants are similar. The higher the value of k_{max}, the better the obtained solutions but the greater the computational effort. For both problems, the configuration with $k_{max} = 5$ achieved the best results in terms of Hypervolume, Coverage, and IGD+. However, the performance increase with respect to $k_{max} = 4$ is the smallest in the comparison, so we did not try higher values of k_{max}. Normally, a larger exploration usually achieves better results, but it is necessary to find a trade-off between the quality of the solutions and the time consumed. Therefore, we selected $k_{max} = 5$ for the following experiments, since it obtained the best results. However, in order to limit the running time of the algorithm, a combined stopping criteria based on the size of the instance should be considered. We detail it later in Section 5.3.3.

Table 5.13 Comparison of the results obtained with (Incremental) and without (Complete) using the incremental evaluation strategy for the MCA problem.

Evaluation	CPUt (s)	PF size	HV	C	IGD+	Spread
Complete	80457.50	748.10	0.2903	0.0000	0.0000	0.5665
Incremental	21428.59	748.10	0.2903	0.0000	0.0000	0.5665

Influence of the incremental evaluation

In this section, we conduct an experiment devoted to analyze the impact of using the efficient evaluation strategy, described in Section 4.6.1, on the performance of the MO-GVNS method. In particular, we configure two variants of the MO-GVNS method: one uses the advanced strategy and the other does not. Again, the rest of the configuration is the same for both variants compared: the first shake procedure, Shake 1, is used; the MO-VND component explores the neighborhoods N_1, N_3, N_2, and N_4, in this order; the stopping criterion is set to $k_{max} = 5$; in order to compare the convergence of the proposed method over time with the different shake procedures, we do not set a time limit in this experiment; finally, the objectives are tackled in the following order within the MO-VND component: MQ, cohesion, coupling, number of modules, and number of isolated modules (in the case of MCA) / difference between the maximum and minimum size of modules (in the case of ECA).

In Tables 5.13 and 5.14, we present the results obtained for MCA and ECA. As can be seen, the quality of the solutions obtained is the same for both variants. However, the use of the incremental evaluation considerably reduces the computational time consumed by the MO-GVNS method. In particular, when using the incremental evaluation, the computational time is reduced by 73.37% in the case of MCA and by 67.23% in the case of ECA. Therefore, we include the efficient evaluation strategy in the configuration of the proposed MO-GVNS algorithm.

Influence of the reduction of the size of neighborhood structures

In this section, we conduct an experiment devoted to analyze the impact of using the second advanced strategy, described in Section 4.6.2, on the performance of the MO-GVNS

Table 5.14 Comparison of the results obtained with (Incremental) and without (Complete) using the incremental evaluation strategy for the ECA problem.

Evaluation	CPUt (s)	PF size	HV	C	IGD+	Spread
Complete	402415.48	1801.60	0.2667	0.0000	0.0000	0.5626
Incremental	131874.37	1801.60	0.2667	0.0000	0.0000	0.5626

Table 5.15 Comparison of the results obtained with (Reduced) and without (Complete) reducing the size of the neighborhoods for the MCA problem.

Size of neighborhoods	CPUt (s)	PF size	HV	C	IGD+	Spread
Complete	67193.25	1216.20	**0.3025**	**0.0869**	**0.0014**	**0.5551**
Reduced	21428.59	748.10	0.2903	0.2885	0.0111	0.5665

method. This strategy consists of reducing the size of neighborhood structures. In particular, we configure two variants of the MO-GVNS method: one that uses the advanced strategy and the other that does not. Again, the rest of the configuration is the same for both variants compared: the first shake procedure, Shake 1, is used; the MO-VND component explores the neighborhoods N_1, N_3, N_2, and N_4, in this order; the stopping criterion is set to $k_{max} = 5$; in order to compare the convergence of the proposed method over time with the different shake procedures, we do not set a time limit in this experiment; finally, the objectives are tackled in the following order within the MO-VND component: MQ, cohesion, coupling, number of modules, and number of isolated modules (in the case of MCA) / difference between the maximum and minimum size of modules (in the case of ECA).

In Tables 5.15 and 5.16, we present the results obtained for both MCA and ECA. As can be seen, there are small differences in the quality of the solutions found by each variant. This can be partially explained by the difference in the size of the neighborhoods explored. Since the local search procedures within the MO-GVNS follow a first improvement approach, a different size of the neighborhood structures leads to different search patterns. Regarding the computational effort, the use of the advanced strategy results in a notable reduction of the computing time. In particular, the use of the strategy results in a reduction of 68.11% and 45.13% of the time consumed for MCA and ECA, respectively. Therefore, in light of the results obtained, we configure the MO-GVNS algorithm to use the reduction of the size of the neighborhood structures.

Table 5.16 Comparison of the results obtained with (Reduced) and without (Complete) reducing the size of the neighborhoods for the ECA problem.

Size of neighborhoods	CPUt (s)	PF size	HV	C	IGD+	Spread
Complete	240329.02	2288.50	**0.2737**	**0.1161**	**0.0017**	**0.5504**
Reduced	131874.37	1801.60	0.2667	0.2951	0.0078	0.5626

Contribution of the objectives as guiding functions

Given the nature of the proposed algorithm, the number of objectives considered as guiding functions directly impacts the computational cost of the algorithm, since a new VND is created for each of the objectives of the problem within the MO-VND component. A guiding function is an objective function that is used to guide the search process. Normally, the objective functions used to evaluate the quality of solutions are used as guiding functions. However, there does not need to be an equivalence between both sets. Therefore, it is interesting to analyze the impact of using the objectives proposed in both the MCA and the ECA problems as guiding functions. Notice that in this case, we are not interested in reducing the number of objectives of the problem, but rather the number of objectives considered during the search process as guiding functions.

Recently, Yuan et al. [140] proposed a methodology to reduce the number of objectives in MOPs. In particular, they proposed three different methods. Given a set of non-dominated solutions, obtained by considering a particular set of objective functions, the proposed methods calculate an error rate for each possible subset of objectives. Usually, assuming that the objective functions are in conflict, the fewer the objectives considered, the lesser the number of non-dominated efficient points in the evaluated set (and the higher the error rate). The goal then is to find a trade-off between the error rate and the number of objectives considered (both to be minimized). To calculate the error rate, the authors proposed three different measures, δ, η, and γ, based on the dominance structure of the front and the correlation between the objectives.

In this experiment, we analyze the error rates of all possible subsets of objectives for the MCA and ECA problems. For the calculation of the error rates, we use the Pareto fronts obtained by the proposed MO-GVNS for the preliminary dataset. In Tables 5.17 and 5.18, we present the results obtained for both the MCA and ECA problems, respectively. For each

Table 5.17 Comparison of the average error rates obtained by removing some of the considered objectives in the MCA problem for the evaluation of the solutions.

Considered objectives	Number of objectives	Avg. error rate
1,2,3,4,5	5	<0.00%
1,3,4,5	4	<0.00%
2,3,4,5	4	<0.00%
1,2,3,4	4	21.97%
1,3,4	3	21.97%
2,3,4	3	21.97%
1,2,4,5	4	47.02%
1,4,5	3	47.02%
2,4,5	3	47.02%

Table 5.18 Comparison of the average error rates obtained by removing some of the considered objectives in the ECA problem for the evaluation of the solutions.

Considered objectives	Number of objectives	Avg. error rate
1,2,3,4,6	5	0.00%
1,3,4,6	4	0.00%
2,3,4,6	4	0.00%
1,2,4,6	4	56.47%
1,4,6	3	56.47%
2,4,6	3	56.47%
1,2,3,4	4	62.89%
1,3,4	3	62.89%
2,3,4	3	62.89%

subset of objectives, we report the mean error rate, averaged among the set of instances, and the three error measures δ, η, and γ. The different combinations of objectives studied are sorted in ascending order depending on the average error rate obtained. Moreover, for the sake of brevity, we have cropped the table, showing only the 10 best subsets of objectives (i.e., those with the smallest error rates). As can be observed in Tables 5.17 and 5.18, removing either coupling or cohesion results in an error rate close to 0% for both problems. This can be explained because coupling and cohesion are antagonist objectives (coupling can be calculated as the number of edges minus cohesion). Therefore, they do not seem to be in conflict.

Table 5.19 Comparison of the results obtained by considering different sets of objectives as guiding functions for the MCA problem.

Guiding functions	CPUt (s)	PF size	HV	C	IGD+	Spread
All	21428.59	748.10	**0.2903**	**0.2066**	0.0029	0.5665
All $\setminus$ {Coupling}	17268.77	720.80	0.2893	0.2272	**0.0028**	0.5658
All $\setminus$ {Cohesion}	17243.87	720.80	0.2893	0.2272	**0.0028**	0.5658
All $\setminus$ {Isolated}	17210.76	705.50	0.2867	0.2748	0.0044	0.5706
All $\setminus$ {Coupling, Isolated}	16750.98	703.70	0.2866	0.2673	0.0041	0.5777
All $\setminus$ {Cohesion, Isolated}	16937.45	703.70	0.2866	0.2673	0.0041	0.5777
All $\setminus$ {MQ}	6428.05	325.10	0.2644	0.4510	0.0288	0.5220
All $\setminus$ {Coupling, MQ}	3131.02	287.60	0.2595	0.4859	0.0339	**0.5005**
All $\setminus$ {Cohesion, MQ}	3085.88	287.60	0.2595	0.4859	0.0339	**0.5005**

As mentioned above, the goal is not to reduce the objectives used to evaluate the solutions, but the number of objectives used as guiding functions during the search process. Here, we compare the results obtained by the MO-GVNS method when considering only a subset of the objectives. In particular, we consider the subsets of objectives that obtained the lowest error rates in Tables 5.17 and 5.18. Notice that, regardless of the subset of objectives considered as guiding functions, all objectives are used to evaluate the solutions obtained and the inclusion of efficient points in the Pareto front. In Tables 5.19 and 5.20, we report the results obtained. As can be observed, considering only a subset of objectives as guiding functions results in worse solutions in terms of quality. However, the difference is sometimes very small (e.g., in the third decimal point). For example, not considering coupling, cohesion, and/or the number of isolated modules for the MCA problem results in an almost identical value for the hypervolume indicator. On the contrary, the computational time consumed is reduced up to 21.83%. In the case of MQ, not considering it as a guiding function results in a greater reduction in the quality of the solutions. In the case of ECA, the results are similar. When not considering the coupling or the size difference between the smallest and largest modules as guiding functions, the method achieves a reduction of up to 32.10% in computational time consumed with just a small detriment to the quality of the solutions.

Given the results obtained, to find a balance between quality and computational effort, we configure our algorithm not to consider coupling and the number of isolated modules

Table 5.20 Comparison of the results obtained by considering different sets of objectives as guiding functions for the ECA problem.

Guiding functions	CPUt (s)	PF size	HV	C	IGD+	Spread
All	131874.37	1801.60	**0.2667**	**0.2035**	0.0027	0.5626
All $\setminus$ {Coupling}	99374.69	1779.70	0.2666	0.2340	**0.0021**	0.5608
All $\setminus$ {Cohesion}	98563.21	1779.70	0.2666	0.2340	**0.0021**	0.5608
All $\setminus$ {Diff}	89541.81	1506.10	0.2533	0.2897	0.0097	0.5520
All $\setminus$ {Coupling, Diff}	68402.46	1489.90	0.2531	0.3140	0.0089	0.5551
All $\setminus$ {Cohesion, Diff}	64126.70	1489.90	0.2531	0.3140	0.0089	0.5551
All $\setminus$ {MQ}	24589.92	531.80	0.2377	0.4281	0.0471	0.5599
All $\setminus$ {Coupling, MQ}	18796.65	486.10	0.2347	0.4153	0.0486	**0.5423**
All $\setminus$ {Cohesion, MQ}	19078.99	486.10	0.2347	0.4153	0.0486	**0.5423**

(in the case of MCA) and coupling and the difference in size between the smallest and largest modules (in the case of ECA) as guiding functions during the search process.

5.3 Comparison with the state of the art

In this section, we compare the performance of the best configurations of the proposed algorithmic procedures with the performance of the best methods available in the literature for each problem studied. In particular, the GRASP-VND method is compared with the state of the art in Section 5.3.1, the GVNS method is compared with the state of the art in Section 5.3.2, and the MO-GVNS method is compared with the state of the art in Section 5.3.3.

5.3.1 Comparison of the GRASP-VND procedure with the best methods for the MQ problem

Here, we compare the performance of the proposed GRASP-VND procedure with the performance of the best method in the literature for the MQ problem, a LNS algorithm [104]. The GRASP-VND procedure has been configured in Section 5.2.1. In summary: the value of α is set at random in each iteration; neighborhood structures N_1, N_3, and N_4 are explored in that order; and a stopping criterion of a maximum of 20 iterations per instance

is established. Moreover, the procedure implements two advanced strategies: an incremental evaluation of the objective function and a reduction in the size of the neighborhood structures. Regarding the LNS procedure, we have executed the code as provided by the original authors[1]. Both algorithms have been implemented with Java OpenJDK 15.0.2 and executed in the same environment, a Microsoft Windows 10 Pro 10.0.19042 x64 with an AMD EPYC 7282 @ 2795 MHz with 8 cores and 8 GB RAM.

In Table 5.21, we report the results obtained in the comparison. For each method, we report the average quality of the solutions found (Avg. O.F.), the average deviation from the best solution found by each method to the best solution found by any method (Avg. Dev. (%)), the number of best solutions found (# Best), and the average computing time (Avg. CPUt (s)). Moreover, we divided the results depending on the size of the instances, reporting the results for small, medium, large, and very large instances separately. Finally, the results for all instances are provided for an overall comparison. As can be observed, the GRASP-VND procedure consistently outperforms the LNS methods in terms of average quality, average deviation, and number of best solutions found. That is, GRASP-VND achieves better solutions than LNS. Regarding the computation time consumed by each method, LNS is faster when tackling small, medium, and large instances. However, GRASP-VND is faster when tackling very large instances. Overall, GRASP-VND is faster on average than LNS.

Finally, we performed a Wilcoxon's Signed Rank test over the obtained results to analyze their significance. We applied the two-tailed version of the test with a significance level of 0.01. The null hypothesis is that there is no difference in the quality of the results obtained, while the alternative hypothesis states that the solutions obtained by one of the algorithms are significantly better than the solutions obtained by the other algorithm. Since the test results in a p-value smaller than 0.01, the null hypothesis can be rejected. Thus, according to the test, the difference in quality of the results obtained by the GRASP-VND procedure and the results obtained by the LNS method is statistically significant with $p < 0.01$.

[1] The original code as published by the authors is available at the following URL: `https://bitbucket.org/marlonmoncores/unirio_lns_cms` [104]

Category	Method	Avg. O.F.	Avg. Dev. (%)	# Best	Avg. CPUt (s)
Small	GRASP-VND	**3.4757**	**0.00%**	**64**	0.09
	LNS [104]	3.4603	0.68%	35	**0.02**
Medium	GRASP-VND	**12.9489**	**<0.01%**	**27**	0.40
	LNS [104]	12.9270	0.26%	10	**0.33**
Large	GRASP-VND	**24.9043**	**0.02%**	**14**	28.20
	LNS [104]	24.8868	0.12%	5	**4.95**
Very large	GRASP-VND	**63.1797**	**<0.01%**	**11**	**235.77**
	LNS [104]	63.1029	0.15%	2	338.87
All	GRASP-VND	**15.0611**	**<0.01%**	**116**	**28.95**
	LNS [104]	15.0374	0.44%	52	36.33

Table 5.21 Comparison of the results obtained by the proposed GRASP-VND procedure and the state-of-the-art LNS method [104] for the MQ problem.

5.3.2 Comparison of the GVNS procedure with the best methods for the FCB problem

In this section, we compare the performance of the proposed GVNS method with the best algorithm known in the literature for the FCB problem: the HGA proposed by Lifeng et al. [105]. The comparison is made using a dataset of 124 real software instances, proposed in the literature [104], as described in Section 5.1. Let us summarize the configuration of the GVNS approach as described in the preliminary experiments: the shake procedure explores N_2; the parameter k_{max} is set to 30; the VND is configured to explore the neighborhoods N_4, N_3, and N_1, in that order; the stopping criterion is set to 15 consecutive iterations without improving the best solution found; and the algorithm incorporates two advanced strategies: the efficient evaluation of the objective function, introduced in Section 4.6.1, and the reduction of the size of the neighborhood structures, introduced in Section 4.6.2.

All algorithms compared were executed in the same computing environment: a Microsoft Windows 10 Pro 10.0.19042 x64 operating system, with an AMD EPYC 7282 @ 2795 MHz CPU with 8 cores and 8 GB RAM. The proposed method was implemented in Java 17.0.1 and using the Metaheuristic Optimization framewoRK (MORK) project [90]. Unfortunately, the original implementation of the HGA algorithm is not publicly available. Instead, we implemented it as described by the original authors [105]. In particular,

we implemented it twice: in Java 17.0.1, the same platform used to implement the GVNS approach; and in Matlab (R2021b Update 1), where the authors of the HGA originally implemented it. Interestingly, we found that our Matlab implementation was more efficient than our Java implementation, since it took advantage of the fast calculation of matrix operations available on the platform, which is an important issue in the design of the algorithm. Therefore, in the comparison, we used our Matlab implementation.

In Table 5.22, we report the results obtained. We present the results obtained for all instances (All (124)) and divided into groups according to the size of the instances, following the same distribution originally given by Marlon et al. [104]: instances with fewer than 79 vertices (Small), instances with fewer than 190 vertices (Medium), instances with fewer than 400 vertices (Large), and instances with more than 400 vertices (Very large). For each group of instances and algorithmic approach compared, we report the average quality of the solutions found (Avg. O.F.), the average deviation from the best solution found in this experiment for each instance by any of the methods compared (Avg. Dev.), the number of instances for which the obtained solution was better or equal to the solution obtained by the other method (# Best), and the average execution time consumed (Avg. CPUt (s)). As can be observed, the GVNS approach obtained solutions of better quality, with less than a 0.01 % of deviation to the best solution found for each instance. In contrast, the solutions found by the HGA had an average deviation of 32.96 %. Moreover, the GVNS obtained the best results for all the instances, whereas the HGA method only obtained the best solutions for 12 of the smallest instances in the dataset. Finally, it can be seen that GVNS was three orders of magnitude faster than HGA. According to the Wilcoxon's signed rank test, the results are statistically significant with $p < 0.001$.

5.3.3 Comparison of the MO-GVNS procedure with the best methods for the MCA and ECA problems

In this section, we compare the performance of the proposed MO-GVNS method with the best algorithms known in the literature for the MCA and ECA problems: a TA-ABC recently proposed to tackle the MCA and ECA problems [8]; the NSGA-III [31]; the Modified Pareto Envelop-Based Selection Algorithm (PESA2) [27]; and the Multi-Objective

Size of instances	Method	Avg. O.F.	Avg. Dev. (%)	# Best	Avg. CPUt (s)
Small (64)	GVNS	**0.6448**	**0.00%**	**64**	**4.29**
	HGA [105]	0.7234	14.10%	12	27.56
Medium (29)	GVNS	**0.5312**	**0.00%**	**29**	**23.89**
	HGA [105]	0.7215	46.67%	0	538.31
Large (18)	GVNS	**0.5075**	**0.00%**	**18**	**103.73**
	HGA [105]	0.7555	54.05%	0	7,629.85
Very large (13)	GVNS	**0.4901**	**0.00%**	**13**	**1084.09**
	HGA [105]	0.7842	65.97%	0	254,173.61
All (124)	GVNS	**0.5821**	**0.00%**	**124**	**136.51**
	HGA [105]	0.7340	32.96%	12	27,894.91

Table 5.22 Comparison of the results obtained with the method proposed in this research, GVNS, and the best known algorithm, HGA [105] for the FCB problem.

Evolutionary Algorithm based on Decomposition (MOEA/D) [147]. The comparison is made using a dataset of 124 real software instances, proposed in the literature [104], as described in Section 5.1. The configuration of the MO-GVNS approach, as described in the preliminary experiments, is the following: the first shake procedure (Shake 1) is used; the MO-VND component explores the neighborhoods N_1, N_3, N_2, and N_4, in this order; the stopping criterion is set to $k_{max} = 5$; the method stops if it has reached a time of four times the number of vertices of the instance at hand ($t_{max} = 4 \cdot |V|$); the objectives are tackled in the following order within the MO-VND component for both MCA and ECA: MQ, cohesion, and the number of modules. Finally, the algorithm incorporates two advanced strategies: the efficient evaluation of the objective functions, introduced in Section 4.6.1, and the reduction of the size of the neighborhood structures, introduced in Section 4.6.2.

In Table 5.23 and Table 5.24, we report the results obtained for both MCA and ECA, respectively. As can be observed, the MO-GVNS method is able to obtain better solutions than the other methods according to all the indicators reported. That is, the solutions obtained by the MO-GVNS method achieve better convergence to the approximate reference set and better distribution along the objective space. For some indicators, such as HV, C, and IGD+, the difference is an order of magnitude. Moreover, the MO-GVNS method is the

Method	CPUt (s)	PF size	HV	C	IGD+	Spread
MO-GVNS	**311.18**	**507.60**	**0.2213**	**0.0264**	**0.0443**	**0.5175**
MOEA/D [147]	336.43	300.00	0.0982	0.5254	0.2184	0.6844
NSGA-III [31]	596.41	479.56	0.0937	0.2719	0.2587	0.5891
PESA2 [27]	643.16	99.63	0.0604	0.7335	0.3209	0.6994
TA-ABC [8]	1037.31	97.70	0.0277	0.2145	0.3991	0.8026

Table 5.23 Comparison of the proposed MO-GVNS with several state-of-the-art methods for the MCA problem.

Method	CPUt (s)	PF size	HV	C	IGD+	Spread
MO-GVNS	**311.16**	**520.64**	**0.1939**	**0.0122**	**0.0180**	**0.5614**
MOEA/D [147]	345.06	300.00	0.0724	0.8010	0.3312	0.6032
NSGA-III [31]	745.81	209.56	0.0889	0.5918	0.3690	0.6777
PESA2 [27]	408.69	89.59	0.0443	0.7386	0.4777	0.7599
TA-ABC [8]	914.74	30.66	0.0284	0.3051	0.5132	0.8932

Table 5.24 Comparison of the proposed MO-GVNS with several state-of-the-art methods for the ECA problem.

fastest one, with an average of 311.18 seconds per instance, and the returned Pareto front contains an order of magnitude more efficient points than the methods under comparison.

Chapter 6

Conclusions and future work

In this doctoral thesis, several problems proposed in the literature for the SMCP have been studied. For each problem, we have proposed different heuristic algorithms and favorably compared them with the best methods available in the state of the art. In Section 6.1, we present the conclusions obtained in this doctoral thesis. Then, in Section 6.2, we outline some open lines of future work that have been identified. Finally, in Section 6.3, we describe the contributions made during the development of this doctoral thesis.

6.1 Conclusions

In the area of SBSE, the use of population-based algorithms is greatly extended and has been shown to be efficient in this context. On the contrary, trajectory-based methods have been little explored. Nevertheless, the use of trajectory-based algorithms, such as GRASP or VNS, has been shown to be efficient for the SMCP. By leveraging some domain knowledge in the design, trajectory-based algorithms can be used to obtain high-quality solutions in very short computational times. In this doctoral thesis, three trajectory-based algorithms have been proposed and compared favorably with the state-of-the-art methods available for the SMCP, studying different problems. In particular, a GRASP-VND has been proposed for the TurboMQ problem, a GVNS has been proposed for the FCB problem, and a MO-GVNS has been proposed for the MCA and ECA problems.

The favorable results obtained support the hypothesis enunciated in this doctoral thesis.

That is, that trajectory-based metaheuristics can obtain better results than population-based methods for the optimization of the structure of software systems in terms of maintainability. Furthermore, due to the general nature of the proposed strategies, we believe that they can be adapted to improve the results of other problems, especially those related to the SMCP. Moreover, several objectives were formulated to guide the research towards the enunciated hypothesis. All the objectives formulated in this doctoral thesis have been achieved. A review of the relevant literature is discussed in Chapter 3 of this doctoral thesis. A definition of the problem and the representation of solutions is given in Chapter 2. The dataset collected and used for the experiments is described in Section 5.1 and in Appendix A. The reference state-of-the-art algorithms have been implemented and are included in the comparisons presented in Section 5.3. The algorithmic procedures proposed in this doctoral thesis are presented in Chapter 4. The configuration and behavior of these methods are described in Section 5.2. The results and their analysis are presented in Section 5.3. Finally, the contributions made during the development of this doctoral thesis are described in Section 6.3. In summary, it seems that all the partial objectives have been achieved, and that the results of this research have been of interest to the scientific community, since they have been disseminated in different scientific venues.

Regarding the proposal of advanced strategies for the SMCP, four contributions of this doctoral thesis can be outlined. An important contribution of this doctoral thesis is the study and categorization of neighborhood structures for the SMCP problems. We realized that neighborhood structures can be classified into one of three categories: neighborhoods defined by operations that do not alter the number of modules; neighborhoods defined by operations that increase the number of modules; and neighborhoods defined by operations that reduce the number of modules. By including at least one neighborhood structure from each category in the design of the algorithms, enough flexibility is ensured for the method to successfully improve the initial solutions.

The second contribution made in this doctoral thesis in relation to advanced strategies is the efficient evaluation of the quality metrics studied. Due to the nature of trajectory-based algorithms, a partial evaluation of only the modified structure of the solution results in a great reduction of the computing time. In this doctoral thesis, an efficient evaluation has been proposed for each objective function studied. Moreover, the impact of these strategies

has been shown to be very beneficial in the preliminary experiments.

The third of the contributions made in relation to advanced strategies is the reduction in the size of neighborhood structures. There exist many moves in a search process that do not improve a solution. In problems where the search space is very large, it is important to identify promising areas to accelerate the search process. As shown in preliminary experiments, the computing time of the proposed algorithms is greatly reduced by identifying promising regions of the search space.

Finally, the fourth contribution in relation to advanced strategies is the analysis of guiding functions. Frequently, the objective functions of an optimization problem are used as guiding functions. However, no correspondence is necessary between the objective and the guiding functions. Moreover, in MOPs, where several objective functions are considered, using only a subset of them as guiding functions might result in a reduction of the computing time with little loss of quality in the solutions obtained. In this doctoral thesis, we have leveraged three methods originally proposed in the literature to reduce the number of objective functions in MOPs to reduce the number of guiding functions. It should be noted that the number of objective functions used to evaluate the solution is not reduced, but only the guiding functions used during the search process.

In this doctoral thesis, we have studied four different variants of the SMCP: TurboMQ, FCB, MCA, and ECA. The first two problems studied are mono-objective. These objective functions have been largely studied in the literature and regarded as good quality metrics to guide the optimization of the modularity of software projects. However, some concerns have been raised regarding their ability to capture the different preferences of software developers. Therefore, the study of software quality optimization through a multi-objective optimization approach seems more suitable, since it allows the stakeholders to introduce their subjective experience in the process. In particular, it allows developers to select an organization among a set of high-quality solutions according to their personal preferences.

6.2 Future work

As mentioned above, trajectory-based algorithms have been little explored in the SBSE literature in contrast to population-based approaches. Accordingly, in this doctoral thesis,

we have focused on the design of trajectory-based methods. However, we believe that a combination of both types of metaheuristic frameworks for the SMCP is worth considering. In this sense, trajectory-based methods could be used to ensure the intensification role, additionally involving some domain knowledge in the design to perform an efficient search. Similarly, exact methods have been little explored. Although the NP nature of the problem makes it difficult to apply exact methods in practice, their combination with search algorithms in matheuristic approaches is worth studying.

In relation to the study of other algorithmic approaches for the SMCP, it would be interesting to analyze the performance of state-of-the-art algorithms that have been proposed for different problems in the SMCP literature. This analysis could help identify common parts and differences between different quality metrics, in addition to possible unexplored transference of search strategies between different variants of the SMCP.

An additional line of future work is the integration of the algorithms proposed in the SDLC of software projects. The small computing time shown by the proposed methods allows them to be easily incorporated into integrated development environments. In this context, the proposed methods could help software developers by providing them with suggestions in real time. This would help developers improve the software quality of their projects, either by implementing the suggested remodularizations or by learning from them. Moreover, these methods could be implemented in software repositories as part of a quality gate to ensure that a quality threshold of the contributed software is maintained.

In order to integrate the proposed approaches into the SDLC of software projects, we believe that there is still work to be done in the search for the optimal set of objectives that reflect the needs of software developers. Specifically, a balance needs to be maintained between the improvement of the quality of the code and the magnitude of the reorganization. This balance is necessary to avoid losing experience and familiarity of software developers with the project at hand. In this sense, it would be beneficial to introduce the minimization of the number of changes to perform as part of a multi-objective approach.

Finally, following the directions given in the related literature [16], we believe that SBSE can be used effectively as a learning tool to investigate common concepts in SE. Accordingly, we consider that the proposed methods, by providing better solutions than the best available algorithms in the state of the art, can allow software practitioners and

researchers to thoughtfully inspect the concept of modularity and the objective functions studied.

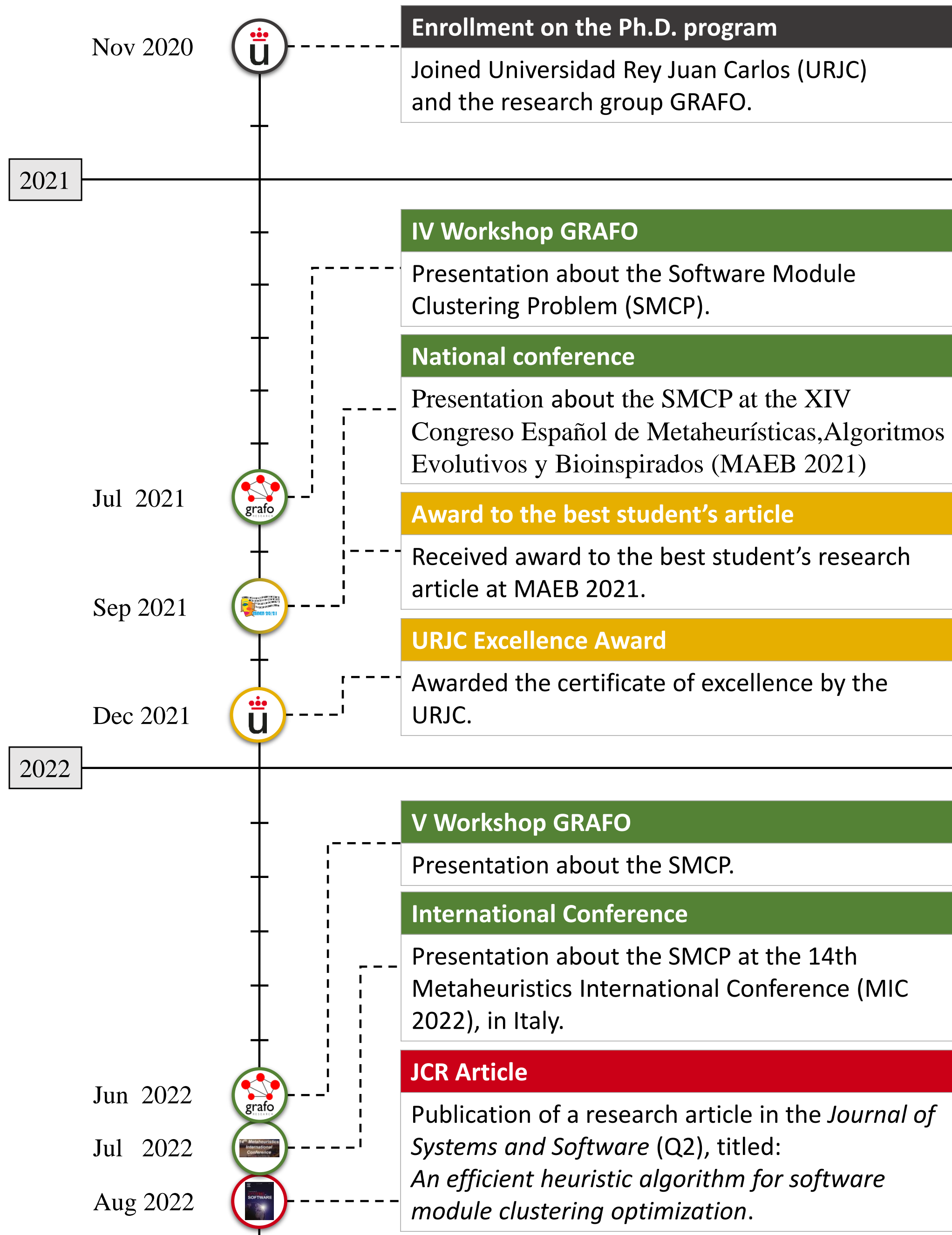
6.3 Contributions

During the development of this doctoral thesis, several publications and presentations have been made in scientific journals and venues. In Figure 6.1, we present a chronological summary of the most relevant events related to this doctoral thesis. Each event is highlighted with a different color, according to the following classification: the publication of research articles in journals ranked in the Journal Citation Reports (JCR)[1] or the Scientific Journal Rankings (SJR)[2] are highlighted in red (■); presentations at conferences and workshops are highlighted in green (■); awards are highlighted in yellow (■); research stays are highlighted in purple (■); and events marking the beginning or end of this doctoral thesis are highlighted in dark gray (■).

As can be seen, the development of this doctoral thesis started in November 2020, when the doctoral candidate joined the Ph.D. program of the Universidad Rey Juan Carlos (URJC) and also joined the Group for Research in Algorithms For Optimization (GRAFO). Then, in 2021, some preliminary findings were presented in the IV Workshop GRAFO and in the national conference XIV Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB) [142], which was held in Málaga, Spain. The research work presented in the latter was awarded the best research article of the conference in the student category. As a result, the URJC Excellence Award was also obtained later that year.

In 2022, part of the research was presented at the V Workshop GRAFO and at the 14th Metaheuristics International Conference (MIC), held in Ortigia-Syracuse, Italy. The latter was published in Lecture Notes in Computer Science (LNCS), ranked in the third quartile (Q3) in the SJR [144]. Moreover, a research article was published in the Journal of Systems and Software, ranked in the second quartile (Q2) in JCR, titled: "An efficient heuristic algorithm for software module clustering optimization" [141]. At the end of the

[1] `https://jcr.clarivate.com/jcr/home`
[2] `https://www.scimagojr.com/journalrank.php`



continued on the next page

Figure 6.1 Timeline of the most relevant events associated with this doctoral thesis.

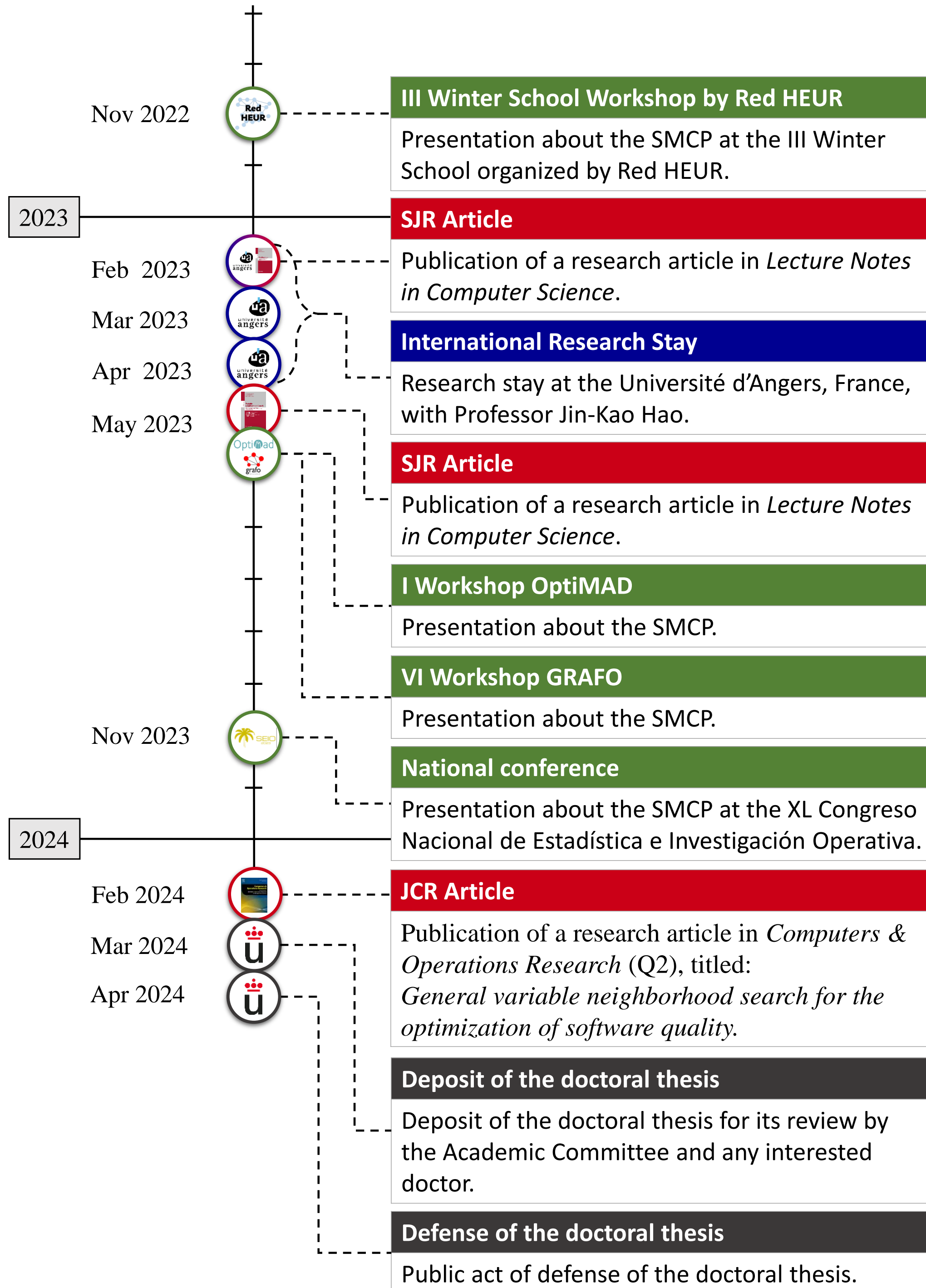


Figure 6.1 Timeline of the most relevant events associated with this doctoral thesis.

year, a presentation was made at the III Winter School Workshop, organized by Red HEUR, in Burgos, Spain.

In 2023, a research article was published in LNCS [143]. In addition, an international research stay was conducted at the Université d'Angers, in Angers, France. During this three-month stay, the doctoral candidate had the honor of collaborating with Professor Jin-Kao Hao. Finally, during 2023, three presentations were made. The first was made at the I Workshop OptiMAD; the second was made at the VI Workshop GRAFO; and the third was made at the national conference XL Congreso Nacional de Estadística e Investigación Operativa, held in Elche, Spain.

As can be observed, 2024 marks the end period of this doctoral thesis. During the first semester of the year, a research article was published in the journal Computers & Operations Research, ranked in the second quartile (Q2) in JCR, titled: "General Variable Neighborhood Search for the optimization of software quality". After that, the manuscript of this doctoral thesis was deposited and the doctoral thesis was defended.

In summary, during the development of this doctoral thesis, several milestones have been achieved, including the following:

- Articles published in journals indexed in the JCR:

  1. J. Yuste, A. Duarte, and E. G. Pardo. An efficient heuristic algorithm for software module clustering optimization. *Journal of Systems and Software*, 190: 111349, 2022.
  2. J. Yuste, E. G. Pardo, and A. Duarte. General Variable Neighborhood Search for the optimization of software quality. *Computers & Operations Research*, page 106584, 2024.
  3. J. Yuste, E. G. Pardo, A. Duarte, and J. Hao. Multi-Objective General Variable Neighborhood Search for Software Maintainability Optimization. *Engineering Applications of Artificial Intelligence*, UNDER REVIEW.

- Articles published in journals indexed in the SJR:

  1. J. Yuste, E. G. Pardo, and A. Duarte. Variable neighborhood descent for software quality optimization. In *Metaheuristics International Conference*, pages 531–536. Springer, 2022.
  2. J. Yuste, E. G. Pardo, and A. Duarte. Multi-objective variable neighborhood search for improving software modularity. In *International Conference on Variable Neighborhood Search*, pages 58–68. Springer, 2022.

- Presentations at international conferences:

  1. J. Yuste, E. G. Pardo, and A. Duarte. Variable neighborhood descent for software quality optimization. In *14th Metaheuristics International Conference*, Ortigia-Syracuse, Italy. 2022, July 11-14.

- Presentations at national conferences:

  1. J. Yuste, E. G. Pardo, and A. Duarte. Heurísticas para la mejora de la mantenibilidad de proyectos software. In *XIX Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA 20/21)*, pages 581–586, 2021.
  2. J. Yuste, E. G. Pardo, A. Duarte, and J. Hao. Optimización multiobjetivo en problemas de calidad de software. In *XL Congreso Nacional de Estadística e Investigación Operativa*, Elche, Spain. 2023, November 7-10.

- Presentations at workshops:

  1. J. Yuste, E. G. Pardo, and A. Duarte. Software Module Clustering Problem. In *IV Workshop GRAFO*, Móstoles, Spain. 2021, July 14-15.
  2. J. Yuste, E. G. Pardo, and A. Duarte. Software Module Clustering Problem. In *V Workshop GRAFO*, Móstoles, Spain. 2022, June 2-3.
  3. J. Yuste, E. G. Pardo, A. Duarte, and J. Hao. Optimización de la calidad de los sistemas software: una aproximación multiobjetivo. In *I OptiMad*, Madrid, Spain. 2023, May 25.

4. J. Yuste, E. G. Pardo, and A. Duarte. Software Module Clustering Problem. In *VI Workshop GRAFO*, Móstoles, Spain. 2023, May 30-31.

- International research stay:

1. Conducted at the Université d'Angers, in Angers, France, under the supervision of Professor Jin-Kao Hao. An article was written as a result of this stay in collaboration with Professor Jin-Kao Hao, which is currently under review in a scientific journal.

Part II

Appendix

Appendix A

Dataset

In this section, we detail the instances contained in the dataset used in the experiments of all the problems studied this doctoral thesis. The dataset is made up of 124 real software instances proposed by previous works [104]. These instances are of varying sizes, having between 2 and 1161 vertices and between 2 and 11722 edges. On average, these instances have 156.37 vertices (with a standard deviation of 216.72) and 948.79 edges (with a standard deviation of 1751.86). In the work where the dataset was proposed for the first time, the instances were divided into four different categories according to their size: 64 small instances (up to 68 vertices), 29 medium instances (from 74 to 182 vertices), 18 large instances (from 190 to 377 vertices), and 13 very large instances (from 413 to 1161 vertices). Following this classification, we present the instances in four different tables: small instances are presented in Table A.1, medium instances are presented in Table A.2, large instances are presented in Table A.3, and very large instances are presented in Table A.4. In each table, we report the number of vertices, the number of edges, and the density of all instances.

Table A.1 Small instances contained in the dataset. These instances have a number of vertices between 2 and 68.

Instance	$\|V\|$	$\|E\|$	Density
squid	2	2	100.00%
small	6	5	16.67%

compiler	13	32	20.51%
random	13	30	19.23%
regexp	14	20	10.99%
jstl	15	20	9.52%
lab4	15	18	8.57%
netkit-ping	15	15	7.14%
nss_ldap	15	16	7.62%
nos	16	52	21.67%
lslayout	17	43	15.81%
boxer	18	29	9.48%
netkit-tftpd	18	23	7.52%
sharutils	19	36	10.53%
mtunis	20	57	15.00%
spdb	21	17	4.05%
xtell	22	57	12.34%
bunch	23	62	12.25%
ispell	24	103	18.66%
netkit-inetd	24	25	4.53%
nanoxml	25	64	10.67%
ciald	26	64	9.85%
jodamoney	26	102	15.69%
Modulizer	26	66	10.15%
bootp	27	75	10.68%
jxlsreader	27	73	10.40%
sysklogd-1	28	74	9.79%
telnetd	28	81	10.71%
crond	29	112	13.79%
netkit-ftp	29	95	11.70%
rcs	29	163	20.07%
seemp	30	61	7.01%
dhcpd-2	31	122	13.12%

cyrus-sasl	32	100	10.08%
tcsh	32	105	10.58%
micq	33	156	14.77%
apache_zip	36	86	6.83%
star	36	89	7.06%
bison	37	179	13.44%
cia	38	185	13.16%
stunnel	38	97	6.90%
minicom	40	257	16.47%
mailx	41	331	20.18%
dot	42	255	14.81%
screen	42	292	16.96%
slang	45	242	12.22%
slrn	45	323	16.31%
net-tools	48	183	8.11%
graph10up49	49	1650	70.15%
wu-ftpd-1	50	230	9.39%
joe	51	540	21.18%
hw	53	51	1.85%
imapd-1	53	298	10.81%
wu-ftpd-3	54	278	9.71%
udt-java	56	227	7.37%
javaocr	58	155	4.69%
dhcpd-1	59	571	16.69%
icecast	60	650	18.36%
pfcda_base	60	197	5.56%
servletapi	61	131	3.58%
php	62	191	5.05%
bunch2	65	151	3.63%
forms	68	270	5.93%

Table A.2 Medium instances contained in the dataset. These instances have a number of vertices between 74 and 182.

Instance	$\|V\|$	$\|E\|$	Density
jscatterplot	74	232	4.29%
jxlscore	79	330	5.36%
elm-2	81	683	10.54%
jfluid	81	315	4.86%
grappa	86	295	4.04%
elm-1	88	941	12.29%
gnupg	88	601	7.85%
inn	90	624	7.79%
bash	92	901	10.76%
jpassword	96	361	3.96%
bitchx	97	1653	17.75%
junit	99	276	2.84%
xntp	111	729	5.97%
acqCIGNA	114	179	1.39%
bunch_2	116	364	2.73%
exim	118	1255	9.09%
xmldom	118	209	1.51%
cia++	124	369	2.42%
tinytim	129	564	3.42%
mod_ssl	135	1095	6.05%
jkaryoscope	136	460	2.51%
ncurses	138	682	3.61%
gae_plugin_core	139	375	1.95%
lynx	148	1745	8.02%
javacc	153	722	3.10%
lucent	153	103	0.44%
JavaGeom	171	1445	4.97%

incl	174	360	1.20%
jdendogram	177	583	1.87%
xmlapi	182	413	1.25%

Table A.3 Large instances contained in the dataset. These instances have a number of vertices between 190 and 377.

Instance	$\|V\|$	$\|E\|$	Density
jmetal	190	1137	3.17%
graph10up193	193	9190	24.80%
dom4j	195	930	2.46%
nmh	198	3262	8.36%
pdf_renderer	199	629	1.60%
Jung_graph_model	207	603	1.41%
jung_visualization	208	919	2.13%
jconsole	220	859	1.78%
pfcda_swing	248	885	1.44%
jml-1.0b4	267	1745	2.46%
jpassword2	269	1348	1.87%
notelab-full	293	1349	1.58%
Poormans_CMS	301	1118	1.24%
log4j	305	1078	1.16%
jtreeview	320	1057	1.04%
bunchall	324	1339	1.28%
JACE	338	1524	1.34%
javaws	377	1403	0.99%

Table A.4 Very large instances contained in the dataset. These instances have a number of vertices between 413 and 1161.

Instance	$\|V\|$	$\|E\|$	Density
swing	413	1513	0.89%
lwjgl-2.8.4	453	1976	0.97%
res_cobol	470	7163	3.25%
ping_libc	481	2854	1.24%
y_base	556	2510	0.81%
krb5	558	3793	1.22%
apache_ant_taskdef	626	2421	0.62%
itextpdf	650	3898	0.92%
apache_lucene_core	738	3726	0.69%
eclipse_jgit	909	5452	0.66%
linux	916	11722	1.40%
apache_ant	1085	5329	0.45%
ylayout	1161	5770	0.43%

Appendix B

Resumen en castellano

Los sistemas *software* son un elemento crucial del día a día en las sociedades modernas. La informática ha revolucionado el mundo en las últimas décadas y es previsible que continúe haciéndolo en el futuro. A medida que los sistemas *software* se vuelven más sofisticados y complejos, comprenderlos se vuelve cada vez más difícil. Además, los sistemas *software* están normalmente en continua evolución, sufriendo modificaciones para adaptarse a las necesidades de los usuarios, añadir nuevas funcionalidades, corregir fallos, etc. Estas continuas modificaciones hacen que los sistemas se deterioren con el tiempo, propiciando la aparición de errores en el código. Según estudios recientes, los sistemas *software* de baja calidad provocaron costes de hasta 2,08 trillones de dólares en 2020, contando únicamente los Estados Unidos de América [77]. Más aun, los errores de código pueden resultar en situaciones catastróficas, como ocurrió en el lanzamiento del satélite Ariane 5 en 1996 [135], el aterrizaje de Mars Polar Lander en 1999 [5] o el error en Starliner en 2019 [92].

Search-Based Software Engineering (SBSE) es un área de investigación cuyo objetivo es resolver algunos de los problemas citados anteriormente. En particular, este campo de investigación se centra en la resolución de tareas de Ingeniería del Software (SE, del inglés *Software Engineering*) abordándolas como problemas de optimización. En este sentido, el objetivo en SBSE es mejorar la calidad de los sistemas *software*.

En esta tesis doctoral, se estudia el *Software Module Clustering Problem* (SMCP), una familia de problemas de optimización enmarcada en el área de SBSE. En particular, se estudian cuatro problemas distintos del SMCP y se proponen tres algoritmos heurísticos para

su resolución. Además, se proponen cuatro estrategias avanzadas para mejorar la eficiencia de los algoritmos propuestos.

La estructura de este apéndice es la siguiente. En la Sección B.1, se presenta la motivación del problema. En la Sección B.2, se describe la metodología seguida. En la Sección B.3, se formulan la hipótesis y los objetivos de esta tesis doctoral. En la Sección B.5, se presenta un resumen del estado del arte del problema. En la Sección B.6, se describen los métodos heurísticos propuestos para resolver los problemas estudiados, así como las estrategias avanzadas diseñadas. En la Sección B.7, se recogen los resultados obtenidos en la comparación de las propuestas algorítmicas con los algoritmos de referencia en el estado del arte. Finalmente, en la Sección B.8, se recogen las conclusiones y los trabajos futuros identificados.

B.1 Motivación

El desarrollo *software* es una tarea compleja. Según un informe reciente sobre proyectos *software* desarrollados entre 2011 y 2015, tan solo el 56% de los proyectos *software* estudiados obtuvieron la satisfacción del cliente [1]. Estos resultados, además, fueron especialmente preocupantes para los proyectos de mayor tamaño, de los que solo entre el 6% y el 11% se completaron con éxito. Según el informe, "la complejidad es una de las principales razones para el fracaso de los proyectos" [1].

El ciclo de vida de un proyecto *software* contiene todas las actividades realizadas para crear o mantener el sistema desde que se concibe hasta que se descataloga [61]. Para realizar código de calidad, es aconsejable seguir una aproximación estructurada, conocida como Ciclo de Vida del Desarrollo Software (SDLC, del inglés *Software Development Life-Cycle*). Un SDLC es un marco de trabajo donde se definen las etapas, actividades y procesos relacionados con el ciclo de vida de un proyecto *software*, su ordenación y los criterios de transición entre los mismos [61]. En un SDLC, se pueden diferenciar dos etapas de manera muy generalizada: el desarrollo del *software* y su mantenimiento. En la etapa de desarrollo, los requisitos se transforman en acciones que crean un elemento del sistema [61]. El mantenimiento *software*, por otra parte, se centra en proveer al sistema de un soporte efectivo para que continúe dando servicio [61]. Esto incluye modificaciones

para corregir o prevenir fallos, mejorar el sistema o adaptarlo a cambios en el entorno [13]. Aunque a menudo se estima que la fase de mantenimiento es menos importante que la fase de desarrollo [34, 82], lo cierto es que el mantenimiento es la fase más costosa de un SDLC, llegando a alcanzar el 80% de los costes totales del proyecto [22]. Además, la mayor parte de los esfuerzos en esta fase se dedican a la comprensión del código ya existente [103].

Según la *International Organization for Standardization* (ISO), la calidad de un sistema *software* se define por ocho atributos: funcionalidad, fiabilidad, eficiencia, usabilidad, seguridad, compatibilidad, mantenibilidad y portabilidad [64]. En este contexto, la mantenibilidad se define como "el grado de efectividad y eficiencia con el que un producto o sistema puede modificarse" [64]. Del mismo modo, la mantenibilidad de los proyectos se define por cinco características relacionadas entre sí: modularidad, reusabilidad, analizabilidad, modificabilidad y testabilidad. Dado que la mantenibilidad de un sistema afecta a la facilidad con la que se podrá modificar a lo largo del tiempo, es un aspecto crucial para el éxito a largo plazo de los sistemas *software* [42].

Uno de los principales problemas a la hora de mantener un sistema es comprenderlo [23, 103]. Históricamente, el *software* se ha dividido en componentes para facilitar la comprensión y modificación de cada componente de manera independiente. No obstante, conforme los sistemas han ido creciendo, ha surgido la necesidad de organizar estos componentes en módulos o paquetes [6]. Esta organización, sin embargo, no se realiza de manera aleatoria, sino teniendo en cuenta el concepto de modularidad. En este contexto, la modularidad se define como el conjunto de "*atributos software que proveen a la estructura de componentes altamente independientes*" [63]. En una estructura modular, los componentes en un mismo módulo están muy relacionados entre sí (alta cohesión) y poco relacionados con los componentes de otros módulos (bajo acoplamiento). Dependiendo del contexto, es posible encontrar diferentes nociones de lo que es un componente *software* (un fichero, una clase, un paquete, etc.). Del mismo modo, en algunos contextos, los términos "módulo" y "componente" se emplean como sinónimos, ya que su distinción aún no está estandarizada [63]. En este documento utilizaremos el término "componente" para referirnos a elementos individuales (ficheros y clases) y el término "módulo" para referirnos a colecciones de componentes (paquetes o carpetas).

Debido a las relaciones entre modularidad y mantenibilidad con la calidad *software* y

el coste de los sistemas, es de vital importancia para el éxito a largo plazo de los proyectos *software* organizar el código siguiendo una estructura modular. En este contexto, el SMCP es un problema de optimización que busca la optimización de la modularidad *software*. La principal motivación de esta tesis doctoral es reducir los costes de mantenimiento y mejorar la calidad *software* mediante el diseño y la implementación de estrategias heurísticas para el SMCP.

B.2 Metodología

El primer uso recogido del término "investigación" data de 1577, con el significado de "*búsqueda diligente o cuidadosa*" [2]. En general, la investigación se puede definir como el "*trabajo sistemático y creativo realizado para incrementar el conocimiento* [...] *y para encontrar nuevas aplicaciones del conocimiento disponible*" [106]. De cualquier modo, la investigación ha de realizarse siguiendo una metodología que permita realizar un trabajo sistemático. En este sentido, existen varias actividades comúnmente aceptadas como parte de un proceso de investigación: observación, formulación de hipótesis, experimentación, análisis de los datos y extracción de conclusiones. Primero, se realiza una observación, cuyo comportamiento se intenta explicar mediante la formulación de una hipótesis. Después, se diseñan y realizan experimentos para validar o rechazar la hipótesis formulada. Una vez que se han realizado los experimentos, es necesario analizar los datos obtenidos y extraer conclusiones. Normalmente, este proceso es iterativo. El análisis de los datos puede dar lugar a nuevas observaciones y/o hipótesis.

Aunque la metodología de investigación es bien conocida en general, algunos detalles difieren dependiendo del área de estudio. En el área de investigación heurística, dado que es un área experimental, el proceso científico se puede extender como se muestra en la Figura B.1. Como se puede ver en el diagrama de actividad mostrado, se presentan diez actividades distintas. En primer lugar, se realiza un estudio del problema y de sus características (paso 1). Después, se realiza un estudio de los algoritmos e instancias de referencia en el estado del arte (paso 2). A continuación, se formula una hipótesis (paso 3). Seguidamente, se diseña (paso 4) e implementa una propuesta algorítmica (paso 5) para apoyar o refutar la hipótesis. A continuación, se realiza un proceso de experimentación con los algoritmos

implementados. Normalmente, los experimentos computacionales con algoritmos se realizan para analizar el comportamiento del algoritmo o comparar su rendimiento con otros algoritmos de referencia para el mismo problema [14]. El primer tipo de experimentación se realiza en el paso 6, mientras que el segundo se realiza en el paso 8. En ambos casos, tras haber realizado los experimentos, se analizan los datos obtenidos (pasos 7 y 9). Estos análisis pueden llevar a la reformulación de la hipótesis o a la modificación del diseño del algoritmo, dando lugar a un proceso iterativo. Finalmente, los resultados obtenidos deben ser publicados para compartir los hallazgos relevantes con la comunidad científica (paso 10). Además, dada la naturaleza experimental del área, el artículo científico debe contener detalles suficientes para permitir la reproducibilidad de los experimentos.

B.3 Hipótesis y objetivos

En esta tesis doctoral, el objetivo es estudiar el SMCP para diseñar e implementar algoritmos de optimización que mejoren los resultados obtenidos por los métodos del estado del arte para la mejora de la mantenibilidad de proyectos *software*. En este sentido, se formula la siguiente hipótesis:

> "*La estructura de los proyectos* software *se puede mejorar mediante el modelado de la mantenibilidad* software *como un problema de optimización y la implementación de metaheurísticas trayectoriales, que pueden obtener soluciones de mayor calidad que las metaheurísticas poblacionales actualmente presentes en el estado del arte mediante el diseño de estrategias eficientes*".

Para alcanzar el objetivo principal, se enuncian los siguientes objetivos parciales:

- **Revisar la literatura del SMCP**. Es necesario estudiar la literatura disponible sobre el SMCP para: (1) entender el problema y su motivación; (2) identificar líneas de investigación en el área; (3) identificar los algoritmos y las instancias de referencia; y (4) estudiar las diferentes ventajas e inconvenientes de las estrategias propuestas para el problema.

- **Estudiar el modelado del problema**. Un análisis de las características del problema

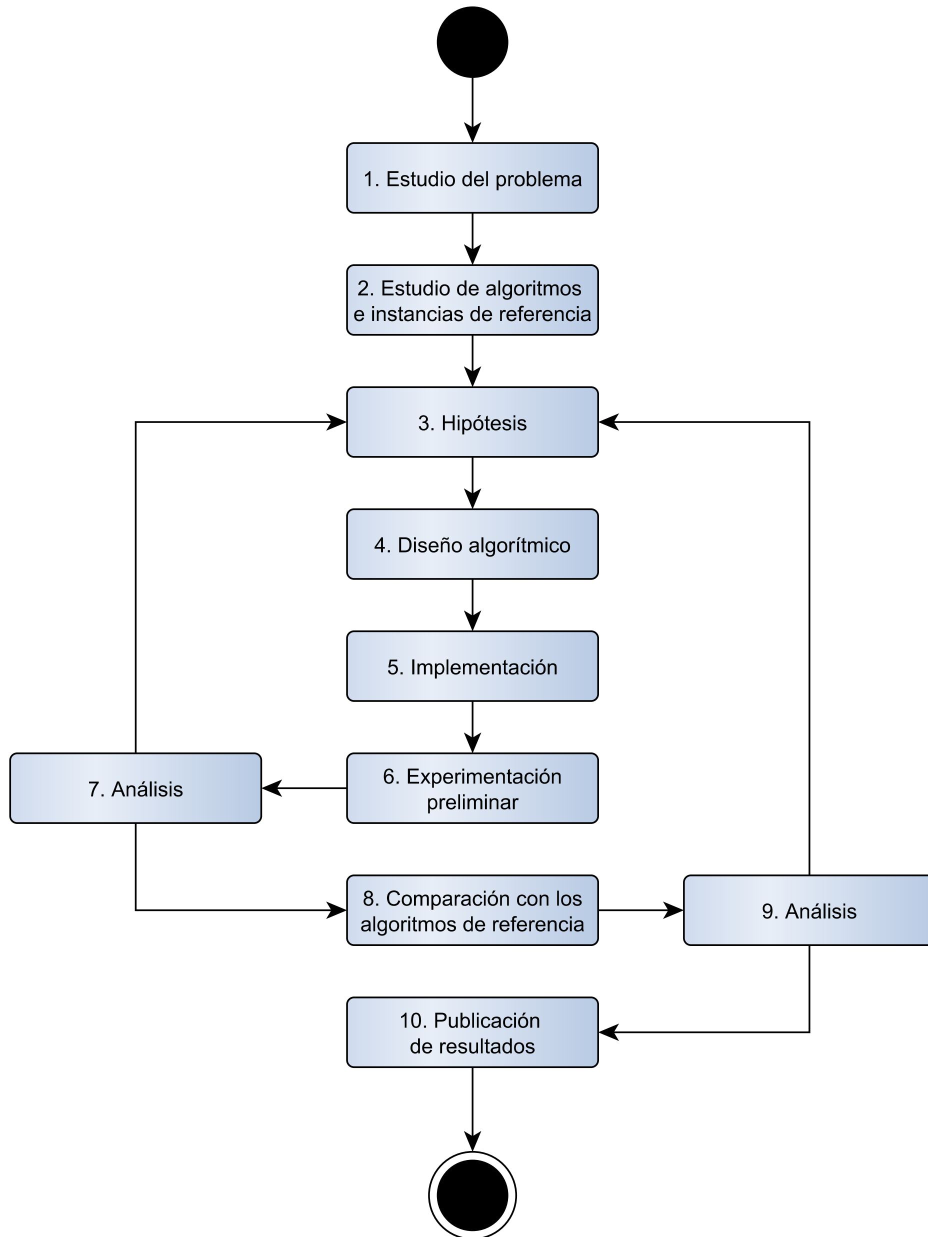


Figura B.1 Diagrama de actividad UML [62] de la metodología seguida durante el desarrollo de esta tesis doctoral.

y la representación de la estructura de los sistemas *software* puede propiciar el diseño de algoritmos y estrategias eficientes para su resolución.

- **Recoger un conjunto relevante de instancias para el problema**. Para poder validar experimentalmente las propuestas algorítmicas y compararlas con los métodos del estado del arte, es importante utilizar un conjunto de instancias aceptadas por la comunidad para el problema. Idealmente, las instancias han de ser reales.

- **Implementar los algoritmos de referencia del estado del arte**. Para poder validar las propuestas algorítmicas, es necesario compararlas con los métodos del estado del arte. Además, para realizar una comparación justa y adecuada, es necesario que todos los algoritmos comparados se ejecuten en el mismo entorno. Por ello, es necesario implementar los algoritmos del estado del arte.

- **Diseñar e implementar métodos basados en metaheurísticas trayectoriales**. Estos algoritmos de optimización se diseñarán teniendo en cuenta las características del problema y sus variantes.

- **Parametrizar los algoritmos propuestos**. En el diseño de los algoritmos, se debe mantener un equilibrio entre la generalización del método para abordar distintos problemas y la especificidad de su diseño para el problema estudiado. Normalmente, los algoritmos heurísticos incluyen ciertos parámetros que pueden ser ajustados para modificar su comportamiento. Estos parámetros, si existen, deben ser analizados y ajustados utilizando un conjunto de instancias representativo de las instancias del problema y distinto (en la medida de lo posible) del conjunto de instancias utilizado en la comparación con otros algoritmos.

- **Analizar el comportamiento de los métodos propuestos**. El comportamiento de los algoritmos propuestos debe ser analizado para estudiar su velocidad de convergencia, su habilidad para escapar de óptimos locales, su robustez, la contribución de los distintos componentes del método a los resultados obtenidos, etc.

- **Comparar los algoritmos propuestos con los mejores métodos disponibles en el estado del arte**. Esta comparación debe realizarse empleando un conjunto relevante

de instancias. Además, debe ser reproducible y realizarse sobre el mismo entorno computacional.

- **Analizar los resultados obtenidos**. Los datos obtenidos de la experimentación deben ser analizados para validar las hipótesis formuladas. Este análisis, además, puede desembocar en experimentos adicionales o mejoras en el diseño de los algoritmos.

- **Documentar el proceso y extraer conclusiones**. El proceso debe ser documentado en detalle para comunicar los hallazgos encontrados a la comunidad científica. Además, el documento resultante tiene que permitir la reproducibilidad de los experimentos.

- **Publicar los resultados tras un proceso de revisión por pares**. Los resultados obtenidos deben enviarse a conferencias y/o revistas relevantes para someterse a un proceso de revisión por pares y, si son aceptables, ser publicados.

B.4 Definición del problema

En los problemas de SMCP, los proyectos *software* se representan normalmente mediante un grafo de dependencias (MDG, del inglés *Module Dependency Graph*). Un MDG es un grafo dirigido con pesos que representa las dependencias entre distintos componentes de un sistema *software*. Formalmente, un MDG se define como un grafo $G = (V, E, W)$, donde V representa el conjunto de vértices del grafo, E representa el conjunto de aristas entre vértices y W representa los pesos de las aristas en E. En este contexto, los vértices representan componentes del sistema *software*, mientras que las aristas representan dependencias entre componentes y los pesos representan la fuerza de esas dependencias. En la Figura B.2, se muestra un MDG de un proyecto *software* ficticio. Como puede observarse, este proyecto tiene diez componentes. El componente uno, representado por el vértice $v1$, tiene dos dependencias con el componente dos, representado por el vértice $v2$. Por ello, la arista que conecta $v1$ con $v2$ tiene asociado un peso igual a dos. De manera similar, se representan las dependencias del resto de componentes del sistema.

Dado un MDG, una solución para un problema de SMCP se representa mediante una

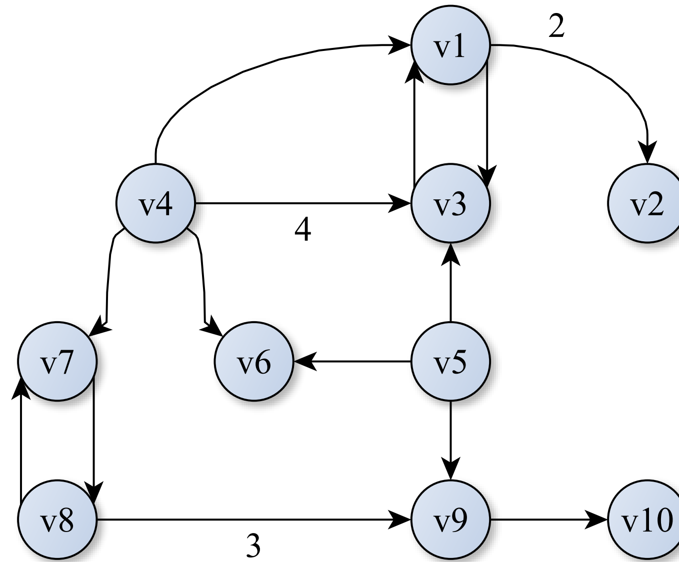


Figura B.2 Representación gráfica de la estructura de un proyecto *software* en un MDG. Por facilidad de lectura, solo se representan los pesos mayores que uno.

agrupación de los vértices del grafo en módulos. Formalmente, una solución es un conjunto $M = \{m_1, m_2, ..., m_n\}$ de subconjuntos disjuntos y no vacíos de vértices en V, donde n representa el número de módulos y $1 \leq n \leq |V|$. En la Figura B.3 se representa una posible solución para el MDG representado en la Figura B.2. Como puede observarse, los componentes se han agrupado en tres módulos: m_1, m_2, y m_3. En particular, $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$ y $m_3 = \{v5, v9, v10\}$.

En la literatura, se han propuesto distintas variantes del SMCP que estudian diferentes métricas de calidad. En las siguientes secciones, se describen algunos de los problemas más relevantes para esta tesis doctoral. En particular, *Modularization Quality* se presenta en la Sección 2.1, *Function of Complexity Balance* se presenta en la Sección 2.2, *Maximizing Cluster Approach* se presenta en la Sección 2.3 y *Equal-size Cluster Approach* se presenta en la Sección 2.4.

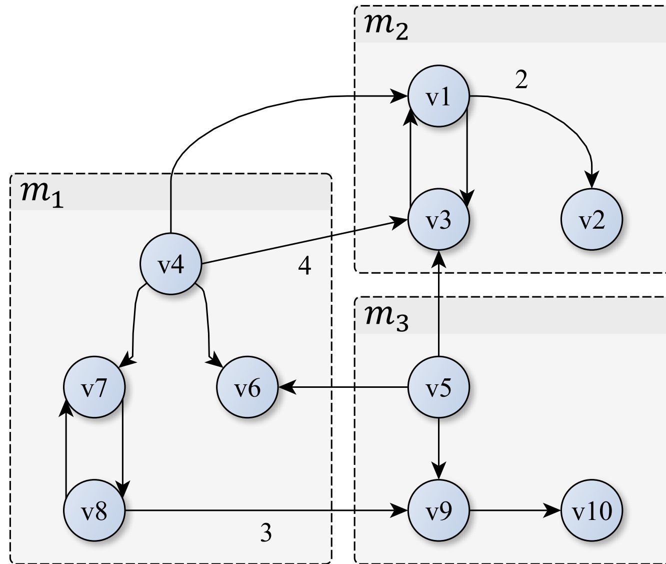


Figura B.3 Representación gráfica de una posible solución para un proyecto *software* ficticio presentado en la Figura B.2, donde los vértices se han dividido en tres módulos: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$.

B.4.1 *Modularization Quality*

Modularization Quality (MQ) es una familia de métricas de calidad propuestas para los problemas del SMCP por primera vez en 1998. En este conjunto de métricas, la calidad de una estructura se mide como un balance entre la cohesión y el acoplamiento de la misma. En particular, se busca maximizar la cohesión y minimizar el acoplamiento. Aunque existen varias funciones objetivo, en esta tesis doctoral nos centramos en la métrica más extendida de la familia MQ: TurboMQ [95]. De aquí en adelante, utilizaremos los términos MQ y TurboMQ indistintamente para referirnos a TurboMQ.

En primer lugar, para calcular el valor de MQ de una solución, es necesario definir los conjuntos de aristas entre vértices pertenecientes a distintos módulos (aristas intermódulo) y los conjuntos de aristas entre vértices pertenecientes a un mismo módulo (aristas intramódulo). Formalmente, las aristas intermódulo entre un par de módulos m_i y m_j se definen de la siguiente manera:

$$Inter(m_i, m_j) = \{(u,v) \in E : u \in m_i \land v \in m_j \bigcup (u,v) \in E : u \in m_j \land v \in m_i\}. \quad \text{(B.1)}$$

De manera similar, las aristas intramódulo de un módulo m_i se definen de la siguiente manera:

$$Intra(m_i) = \{\{u,v\} \in E : u, v \in m_i\}. \quad \text{(B.2)}$$

Una vez definidos los términos de aristas intra- e intermódulo, podemos describir cómo se calcula el valor de MQ de una solución. Para ello, primero es necesario calcular el factor de modularidad de cada módulo de la solución. Este factor, denominado CF, se calcula para cada módulo m_i de la siguiente manera:

$$CF_i = \begin{cases} 0 & \text{si } \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} (\varepsilon_{i,j})} & \text{si } \mu_i \neq 0. \end{cases} \quad \text{(B.3)}$$

donde $\varepsilon_{i,j}$ representa la suma de los pesos de las aristas que conectan vértices pertenecientes al módulos m_i con vértices pertenecientes al módulo m_j y μ_i representa la suma de los pesos de las aristas entre vértices que pertenecen al módulo m_i. Formalmente:

$$\varepsilon_{i,j} = \sum_{(u,v) \in Inter(m_i, m_j)} w(u,v). \quad \text{(B.4)}$$

$$\mu_i = \sum_{u,v \in Intra(m_i)} w(u,v). \quad \text{(B.5)}$$

A continuación, el valor de MQ de una solución se calcula como la suma de los factores de modularidad de los módulos en la solución. Formalmente:

$$\text{TurboMQ} = \sum_{i=1}^{n} CF_i. \quad \text{(B.6)}$$

Como puede observarse, la complejidad de calcular el valor de MQ para una solución es $O(|E|)$. Por último, cabe destacar que el objetivo es maximizar el valor de MQ.

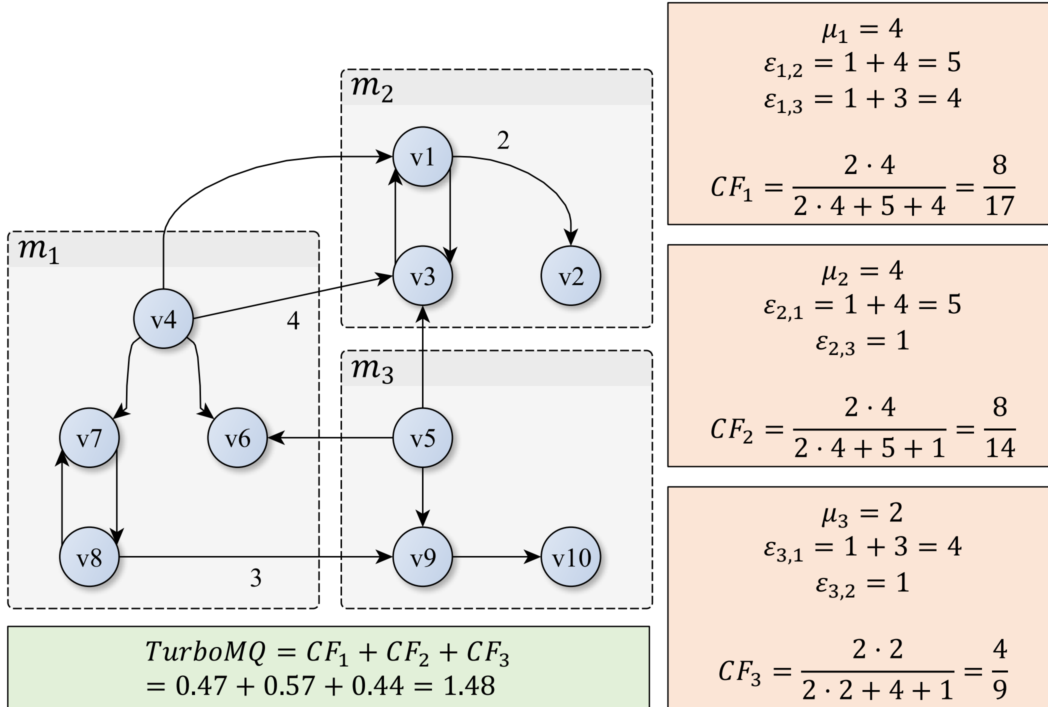


Figura B.4 Cálculo del valor de TurboMQ para una solución para el proyecto presentado en la Figura B.2 con tres módulos: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$, and $m_3 = \{v5, v9, v10\}$.

En la Figura B.4, se representa la evaluación del valor de MQ de la solución representada en la Figura B.3. Al lado de cada módulo, se recogen los valores de intraconectividad e interconectividad del módulo en cuestión, así como su factor de modularidad. Para el módulo m_1, $\mu_1 = 4$, ya que existen cuatro aristas con peso igual a uno que conectan vértices pertenecientes al módulo m_1 $((v4, v6), (v4, v7), (v7, v8)$ y $(v8, v7))$. De manera similar, $\varepsilon_{1,2} = 5$, ya que existe una arista con peso igual a uno que conecta un vértice perteneciente al módulo m_1 con un vértice perteneciente al módulo m_2 $((v4, v1))$ y una arista con peso igual a cuatro que conecta un vértice perteneciente al módulo m_1 con un vértice perteneciente al módulo m_2 $((v4, v3))$. Por ello, $CF_1 = \frac{2 \cdot 4}{2 \cdot 4 + 5 + 4} = \frac{8}{17} = 0.47$. El resto de los factores de modularidad se calculan de manera similar. Finalmente, la calidad de la solución se calcula como TurboMQ $= CF_1 + CF_2 + CF_3 = 1.48$.

B.4.2 *Function of Complexity Balance*

El problema FCB (del inglés *Function of Complexity Balance*) se propuso como una alternativa a MQ con el objetivo de reducir el número de módulos aislados (módulos con un solo vértice o componente) [105]. Formalmente, el valor de FCB de una solución se calcula de la siguiente manera:

$$\text{FCB} = \frac{C + \max\limits_{m_i \,\in\, M}(\mu_i)}{T}, \tag{B.7}$$

donde C representa el acoplamiento de la solución y μ_i representa la cohesión del módulo m_i. En particular, C se calcula de la siguiente manera:

$$C = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \varepsilon_{i,j}. \tag{B.8}$$

Por otro lado, el valor μ_i se calcula como se ha descrito en la Ecuación B.5. Finalmente, T se calcula como la suma de los pesos de todas las aristas de la solución. Como puede observarse, el valor de T es una constante independiente de la organización del grafo en módulos. Formalmente,

$$T = \sum_{e=(u,v) \,\in\, E} w_{u,v}. \tag{B.9}$$

Esta constante se utiliza para normalizar el valor de FCB entre cero y uno, permitiendo la comparación de la calidad de soluciones para distintos proyectos *software*. En el caso de FCB, el objetivo es minimizar el valor de la métrica.

En la Figura B.5, se representa la evaluación de FCB para la solución presentada en la Figura B.3. Al lado de cada módulo, se representa su cohesión. Para el módulo m_1, $\mu_1 = 4$, ya que existen cuatro aristas con peso igual a uno que conectan vértices pertenecientes al módulo m_1 ($(v4, v6), (v4, v7), (v7, v8)$ y $(v8, v7)$). El acoplamiento del proyecto (C) es igual a diez, ya que existen tres aristas con peso igual a uno que unen vértices pertenecientes a módulos distintos ($(v4, v1), (v5, v3)$ y $(v5, v6)$), una con peso igual a cuatro ($(v4, v3)$) y otra con peso igual a tres ($(v8, v9)$). La constante T es igual a veinte, la suma de los

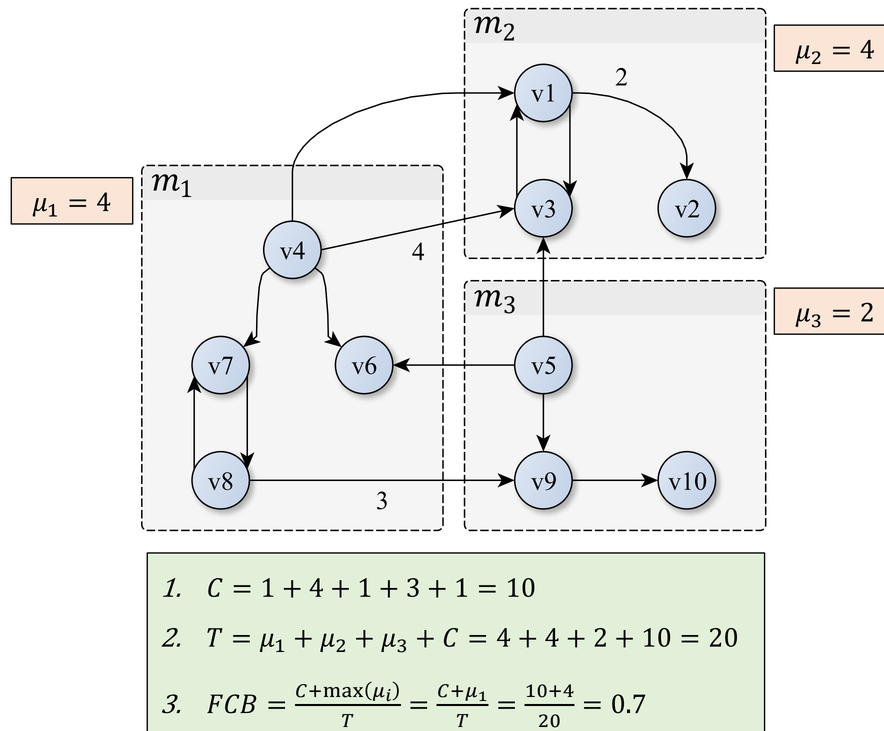


Figura B.5 Cálculo del valor de FCB para una solución para el proyecto presentado en la Figura B.2 con tres módulos: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$ y $m_3 = \{v5, v9, v10\}$.

pesos de todas las aristas. Finalmente, la calidad de la solución se puede calcular como $FCB = \frac{C+max(\mu_i)}{T} = \frac{14}{20} = 0.7$.

B.4.3 *Maximizing Cluster Approach*

Cuando el principio de máxima cohesión y mínimo acoplamiento se lleva al extremo, la mejor solución posible consiste en agrupar todos los componentes en el mismo módulo. Sin embargo, una solución como la descrita es trivial y no deseable. En la práctica, existen

otros objetivos que deben considerarse simultáneamente para la mejora de la mantenibilidad de un proyecto *software*. En 2011, Praditwong et al. plantearon dos problemas multiobjetivo para mejorar la mantenibilidad de proyectos *software* [121]. Esta propuesta perseguía dos ventajas: (1) diferentes objetivos en conflicto reflejan de manera más acertada los deseos de los desarrolladores de *software* y (2) proporcionar un conjunto de soluciones no dominadas a los decisores permite la introducción de la experencia y preferencias de los desarrolladores en el proceso.

El primer problema que plantearon se conoce como MCA (del inglés *Maximizing Cluster Approach*). Este problema considera cinco objetivos:

1. **Acoplamiento**. El primer objetivo consiste en la minimización del acoplamiento del sistema (véase la Ecuación B.8).

2. **Cohesión**. El segundo objetivo consiste en la maximización de la cohesión de la arquitectura. Formalmente, la cohesión se calcula como:

$$\text{Cohesion} = \sum_{i=1}^{n} \mu_i, \tag{B.10}$$

   donde μ_i se calcula como se describe en la ecuación B.5.

3. **TurboMQ**. El tercer objetivo consiste en la maximización de MQ (véase la Ecuación 2.9).

4. **Número de módulos**. El cuarto objetivo consiste en la maximización del número de módulos. Este valor es igual a n.

5. **Número de módulos aislados**. Finalmente, el quinto objetivo considera la minimización del número de módulos aislados (módulos que contienen un solo vértice).

En la Figura B.6, se representa la evaluación de los objetivos de MCA para la solución representada en la Figura B.3. Al lado de cada módulo, se detalla su cohesión. Para el módulo m_1, $\mu_1 = 4$, ya que existen cuatro aristas con peso igual a uno que conectan vértices pertenecientes a m_1 ($(v4,v6),(v4,v7),(v7,v8)$ y $(v8,v7)$). El acoplamiento del proyecto (C) es igual a diez, ya que existen tres aristas con peso igual a uno que unen vértices

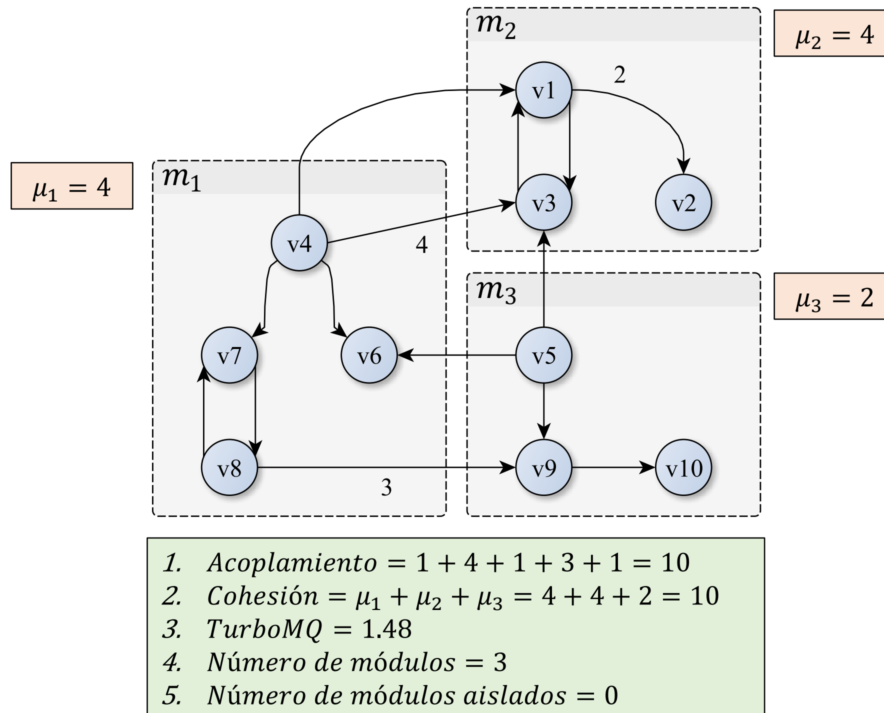


Figura B.6 Cálculo de los valores de los objetivos del problema MCA para una solución para el proyecto presentado en la Figura B.2 con tres módulos: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$ y $m_3 = \{v5, v9, v10\}$.

pertenecientes a módulos distintos ($(v4, v1), (v5, v3)$ y $(v5, v6)$), una con peso igual a cuatro ($(v4, v3)$) y otra con peso igual a tres ($(v8, v9)$). La cohesión de la solución también es igual a diez, ya que $\mu_1 = 4$, $\mu_2 = 4$ y $\mu_3 = 2$. El valor de TurboMQ es igual a 1.48, tal y como se describió en la Figura 2.5. El número de módulos en la solución es igual a tres. Finalmente, como no hay ningún módulo que contenga solo un vértice, el número de módulos aislados es igual a cero.

B.4.4 *Equal-size Cluster Approach*

En el mismo trabajo que proponía MCA, se propuso el problema ECA (del inglés *Equal-size Cluster Approach*). ECA considera cinco objetivos diferentes. Los primeros cuatro son comunes a MCA: acoplamiento, cohesión, TurboMQ y el número de módulos. Sin embargo, el quinto objetivo, la minimización del número de módulos aislados, se sustituye por la minimización de la diferencia de tamaño entre el módulo más grande y el módulo más pequeño de la solución. En este contexto, el tamaño de un módulo es igual al número de vértices pertenecientes a dicho módulo. Debido a las similitudes entre MCA y ECA, estos dos problemas se suelen estudiar juntos.

En la Figura B.7, se calculan los valores de los objetivos de ECA para la solución representada en la Figura B.3. Al lado de cada módulo, se detalla su cohesión. Para el módulo m_1, $\mu_1 = 4$, ya que existen cuatro aristas con peso igual a uno que conectan vértices pertenecientes a m_1 $((v4, v6), (v4, v7), (v7, v8)$ y $(v8, v7))$. El acoplamiento del proyecto (C) es igual a diez, ya que existen tres aristas con peso igual a uno que unen vértices pertenecientes a módulos distintos $((v4, v1), (v5, v3)$ y $(v5, v6))$, una con peso igual a cuatro $((v4, v3))$ y otra con peso igual a tres $((v8, v9))$. La cohesión de la solución también es igual a diez, ya que $\mu_1 = 4$, $\mu_2 = 4$ y $\mu_3 = 2$. El valor de TurboMQ es igual a 1.48, tal y como se describió en la Figura 2.5. El número de módulos en la solución es igual a tres. Finalmente, la diferencia de tamaño entre el módulo más grande (m_1, con cuatro vértices) y el módulo más pequeño (m_3, con tres vértices) es igual a uno.

B.5 Estado del arte

Uno de los primeros usos del término SBSE data de 2001, cuando Mark Harman y Bryan F. Jones afirmaron que SBSE era un campo emergente de investigación y que esperaban "*ver un desarrollo dramático del área*" [54]. Más de 20 años después, en 2023, el Simposio de SBSE (SSBSE, del inglés *Symposium on Search Based Software Engineering*) celebraba su decimoquinta edición [10]. Solo en sus primeras once ediciones, el simposio ya había recibido contribuciones de más de 290 autores de 25 países diferentes [25]. Las contribuciones de la comunidad de SBSE a revistas científicas también ha sido significativa en las

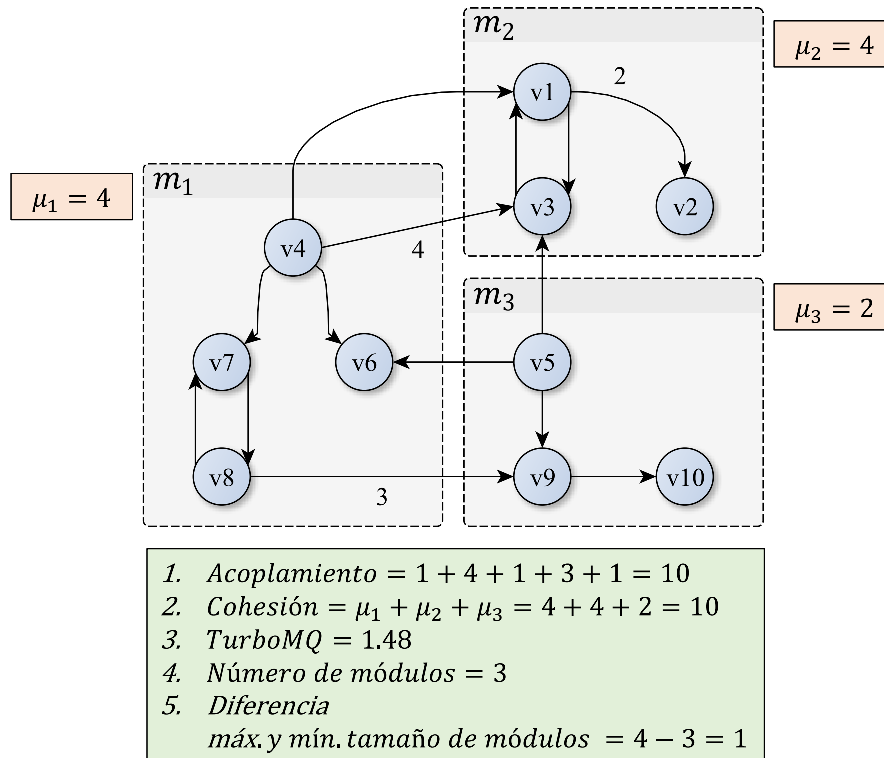


Figura B.7 Cálculo de los valores de los objetivos del problema ECA para una solución para el proyecto presentado en la Figura B.2 con tres módulos: $m_1 = \{v4, v6, v7, v8\}$, $m_2 = \{v1, v2, v3\}$ y $m_3 = \{v5, v9, v10\}$.

últimas décadas. Solo en España, más de 145 autores de más de 19 instituciones diferentes han publicado resultados sobre distintos problemas del área [125].

En el área de SBSE, existen diferentes familias de problemas de optimización que surgen en multitud de actividades de SE a lo largo del SDLC, desde la gestión del proyecto y la priorización de requisitos a la generación de casos de prueba o la corrección automática de código. En este contexto, el SMCP es una familia de problemas de optimización cuyo objetivo es encontrar la mejor organización posible del código de un proyecto *software* en términos de modularidad. En la Tabla B.1 y en la Tabla B.2, se presenta un resumen cronológico de trabajos que han estudiado el SMCP. La primera tabla recoge trabajos que

estudian variantes monoobjetivo del problema, mientras que la segunda tabla recoge trabajos que estudian variantes multiobjetivo del SMCP. Por cada trabajo mostrado en la tabla, se recogen el año de publicación, la referencia al trabajo en cuestión, el problema estudiado y la metaheurística o estrategia en la que se basa el método propuesto en el trabajo. Además, las propuestas basadas en métodos poblacionales se han coloreado con fondo verde, mientras que las propuestas basadas en métodos trayectoriales se han coloreado con fondo rojo.

Como se puede observar en la Tabla B.1, la primera aproximación al SMCP fue propuesta por Mancoridis et al. en 1998 [89]. En el trabajo citado, los autores propusieron un problema conocido como MQ (del inglés *Modularization Quality*). En este problema, la modularidad de un proyecto se mide como un balance entre el acoplamiento y la cohesión de su estructura. Este ha sido el problema más estudiado en el SMCP. Aún así, algunos autores han identificado aspectos mejorables en MQ y han propuesto problemas alternativos, entre los que cabe destacar FCB (del inglés *Function of Complexity Balance*) [105].

Dado que el proceso de modularización realizado por los desarrolladores frecuentemente incluye cierta subjetividad [16], otros autores han preferido abordar el problema desde un enfoque multiobjetivo. En este sentido, un enfoque multiobjetivo ofrece, principalmente, dos ventajas: (1) diferentes objetivos en conflicto pueden reflejar la modularidad de un sistema de manera más acertada que una sola métrica, y (2) la presentación de un conjunto de soluciones de calidad a un decisor (por ejemplo, un desarrollador *software*) puede permitir que los interesados prioricen ciertas métricas sobre otras, dependiendo de sus preferencias. En este contexto, las propuestas más estudiadas y aceptadas en la literatura son las publicadas por Praditwong et al. en 2011 [121]: MCA (del inglés *Maximizing Cluster Approach*) y ECA (del inglés *Equal-size Cluster Approach*).

Independientemente de las variantes estudiadas, existen multitud de propuestas algorítmicas en la literatura para resolver problemas del SMCP. Es importante mencionar que algunos de los problemas de SMCP son NP-completos [18]. Por lo tanto, los métodos exactos no son adecuados salvo para instancias muy pequeñas [95]. En su lugar, los algoritmos aproximados son más convenientes [25, 55, 129]. En general, como se puede observar en la tabla, los métodos poblacionales se han utilizado con mayor frecuencia para resolver estos problemas. Sin embargo, algunos autores han destacado la ausencia de metaheurísticas

trayectoriales eficientes como ILS (del inglés *Iterative Local Search*) o VNS (del inglés *Variable Neighborhood Search*) en la literatura [129]. En este sentido, algunas propuestas recientes han optado por utilizar métodos trayectoriales, obteniendo comparaciones favorables con el estado del arte, como LNS (del inglés *Large Neighborhood Search*) [104], VNS [144, 143] o GRASP (del inglés *Greedy Randomized Adaptive Search Procedure*) [141].

Table B.1 Resumen cronológico de propuestas para problemas monoobjetivo pertenecientes al SMCP.

		Problemas monoobjetivo							
Año	Referencia	MQ	DQCQ	MS	CCCS	MNCC	EOF	FCB	LCC
1998	[89]	HC	-	-	-	-	-	-	-
1999	[88]	HC	-	-	-	-	-	-	-
	[37]	GA	-	-	-	-	-	-	-
2001	[94]	NAHC, SAHC	-	-	-	-	-	-	-
2002	[96]	SAHC	-	-	-	-	-	-	-
2005	[85]	HC, GAHC	-	-	-	-	-	-	-
2006	[97]	GA, HC	-	-	-	-	-	-	-
2008	[98]	SA	-	-	-	-	-	-	-
2009	[3]	-	SA	-	-	-	-	-	-
	[87]	HGA	-	-	-	-	-	-	-
2011	[120]	GA	-	-	-	-	-	-	-
2014	[116]	ILS	-	-	-	-	-	-	-
	[86]	FA	-	-	-	-	-	-	-
2016	[56]	-	-	HC, GA, MAEA	-	-	-	-	-
	[65]	E-CDGM	-	-	-	-	-	-	-
	[68]	GA	-	-	-	-	-	-	-
	[133]	EoD	-	-	-	-	-	-	-
2017	[57]	MAEA	-	-	-	-	-	-	-
	[7]	-	-	-	HS	-	-	-	-
	[73]	HC	-	-	-	-	-	-	-

Table B.1 Resumen cronológico de propuestas para problemas monoobjetivo pertenecientes al SMCP.

		Problemas monoobjetivo							
Año	Referencia	MQ	DQCQ	MS	CCCS	MNCC	EOF	FCB	LCC
2018	[123]	-	-	-	-	PSO	-	-	-
	[104]	LNS	-	-	-	-	-	-	-
2019	[66]	-	-	-	-	-	GA	-	-
2020	[105]	-	-	-	-	-	-	HGA	-
2021	[119]	-	-	-	-	-	-	-	GMA
2022	[141]	GRASP-VND	-	-	-	-	-	-	-
	[144]	-	-	-	-	-	-	VND	-
2024	[145]	-	-	-	-	-	-	GVNS	-

Table B.2 Resumen cronológico de propuestas para problemas multiobjetivo pertenecientes al SMCP.

		Problemas multiobjetivo					
Año	Referencia	MCA, ECA	SSH	MFMC	IFF	E-MCA, E-ECA	MOF
2011	[121]	GA	-	-	-	-	-
2015	[99]	-	NSGA-III	-	-	-	-
2016	[78]	MHypEA	-	-	-	-	-
2017	[59]	-	-	HC	-	-	-
2018	[128]	-	-	-	IEC	-	-
	[8]	TA-ABC	-	-	-	-	-
	[24]	-	-	-	-	MaABC	-
2019	[67]	-	-	-	-	-	EoD
2022	[9]	GA	-	-	-	-	-
	[122]	-	-	-	-	GLMPSO	-
	[143]	MO-VND	-	-	-	-	-

B.6 Propuestas algorítmicas

En esta tesis doctoral, se proponen tres algoritmos para problemas del SMCP. El primer algoritmo, diseñado para el problema MQ, está basado en la metodología GRASP (del inglés *Greedy Randomized Adaptive Search Procedure*) [43, 44], donde la búsqueda local se sustituye por un procedimiento VND (del inglés *Variable Neighborhood Descent*) [50]. El método consta de tres componentes: un preprocesado de las instancias, un procedimiento constructivo y un procedimiento VND. En primer lugar, el algoritmo realiza un preprocesado de la instancia, en el que el tamaño del grafo de entrada se reduce. La calidad del grafo resultante es igual que la del grafo de entrada. A continuación, el algoritmo procede a construir y mejorar soluciones. En el procedimiento constructivo, el método comienza generando una solución vacía. De manera iterativa, se procede a añadir vértices a la solución anterior utilizando un mecanismo semivoraz. Para ello, se propone una función voraz que evalúa la proximidad de los vértices a la solución parcial en cada iteración. Finalmente, la solución inicial obtenida por el procedimiento constructivo se mejora en un esquema VND, que explora tres vecindades distintas. El método propuesto es un método multiarranque. Por lo tanto, el proceso de construcción y mejora de soluciones se lleva a cabo repetidas veces. Una vez que termina, el método GRASP-VND devuelve la mejor solución encontrada a lo largo del proceso de búsqueda.

El segundo método propuesto, diseñado para el problema FCB, se basa en la metodología GVNS (del inglés *General Variable Neighborhood Search*) [50]. Este método consta de dos componentes: un mecanismo de diversificación llamado *shake* y un procedimiento de mejora VND. El mecanismo *shake* introduce una perturbación en una solución dada. El objetivo de este mecanismo es escapar de óptimos locales para diversificar la búsqueda en el espacio de soluciones. Cuantas más iteraciones se realizan sin mejorar la mejor solución encontrada, mayor es la magnitud de la perturbación introducida por el procedimiento *shake*. Finalmente, para construir la solución inicial, se propone un constructivo aleatorio.

El tercer método propuesto, diseñado para los problemas MCA y ECA, se basa en la metodología MO-VNS (del inglés *Multi-Objective Variable Neighborhood Search*) [39]. Esta metodología extiende las ideas de Variable Neighborhood Search (VNS) [100] para abordar problemas multiobjetivo. En particular, se propone un procedimiento basado en

MO-GVNS (del inglés *Multi-Objective General Variable Neighborhood Search*). Para la construcción de la solución inicial, se propone un constructivo aglomerativo basado en las ideas de *Path Relinking* [52]. Este constructivo genera un conjunto de puntos eficientes no dominados entre sí al recorrer el espacio de objetivos entre dos soluciones triviales. El procedimiento constructivo, se puede extender a otras funciones voraces distintas de la utilizada y además se puede extender para utilizar un criterio semivoraz. Para el procedimiento de perturbación de las soluciones (MO-*Shake*), se proponen cuatro métodos distintos, combinando estrategias voraces y aleatorias. Finalmente, las soluciones se mejoran mediante un procedimiento MO-VND, que extiende el esquema VND a problemas multiobjetivo.

Todos los métodos anteriormente propuestos incluyen un componente VND o MO-VND que explora de manera sistemática un conjunto de estructuras de vecindad para mejorar una solución dada. En esta tesis doctoral, además, se proponen seis estructuras de vecindad distintas, basadas en operaciones de inserción (N_1), intercambio (N_2), extracción (N_3), destrucción (N_4), combinación (N_5) y división (N_6).

Para mejorar la eficiencia de las propuestas algorítmicas, se proponen cuatro estrategias avanzadas. En primer lugar, se realiza una categorización de las estructuras de vecindad para el problema. En particular, se han identificado tres categorías de vecindades: (1) vecindades cuyas operaciones no están diseñadas para alterar el número de módulos de la solución, (2) vecindades cuyas operaciones están diseñadas para aumentar el número de módulos en la solución y (3) vecindades cuyas operaciones están diseñadas para reducir el número de módulos en la solución. Siguiendo esta clasificación, las vecindades N_1 y N_2 pertenecen a la primera categoría, las vecindades N_3 y N_6 pertenecen a la segunda categoría y las vecindades N_4 y N_5 pertenecen a la tercera categoría. La utilización de al menos una vecindad de cada categoría dota a los algoritmos propuestos de flexibilidad para aumentar, disminuir o mantener el número de módulos en una solución dada.

La segunda estrategia propuesta consiste en una evaluación eficiente de las funciones objetivo. Dado que las metaheurísticas trayectoriales se basan en la mejora gradual de una solución o conjunto de soluciones mediante pequeñas modificaciones, una factorización de la función objetivo para evaluar parcialmente las soluciones reduce considerablemente el tiempo de cómputo de la calidad de estas.

La tercera estrategia propuesta consiste en identificar regiones prometedoras del espacio de búsqueda. Como consecuencia, se reduce el tamaño de las estructuras de vecindad propuestas, disminuyendo el esfuerzo computacional de los algoritmos propuestos al evitar explorar soluciones no prometedoras.

La cuarta y última estrategia propuesta en este tesis doctoral consiste en analizar la contribución de las funciones guía utilizadas durante el proceso de búsqueda. Dado que el procedimiento MO-GVNS propuesto mejora una solución considerando cada objetivo por separado, la reducción del número de objetivos explorados en el proceso de búsqueda reduce el tiempo computacional empleado por el algoritmo. No obstante, la reducción de las funciones guía también implica una reducción de la calidad de las soluciones obtenidas. Por ello, se estudia la reducción del número de funciones guía para encontrar un equilibrio entre coste computacional y calidad de las soluciones. Es importante destacar que lo que se busca es reducir el número de funciones guía, no el número de funciones objetivo utilizadas para evaluar las soluciones.

B.7 Resultados

Para evaluar las propuestas algorítmicas descritas en la sección anterior, se ha empleado un conjunto de 124 instancias de proyectos *software* reales. Estas instancias han sido recogidas y empleadas anteriormente en la literatura para el SMCP. En el conjunto de instancias, se pueden encontrar grafos con tamaños dispares, desde grafos con 2 vértices y 2 aristas hasta grafos con 1161 vértices y 5770 aristas. De media, las instancias tienen 156.37 vértices y 948.79 aristas. En el trabajo en el que este conjunto de instancias se propuso por primera vez, las instancias se dividieron en cuatro grupos dependiendo de su tamaño: 64 instancias pequeñas (de hasta 68 vértices), 29 instancias medianas (de 74 a 182 vértices), 18 instancias grandes (de 190 a 377 vértices) y 13 instancias muy grandes (de 413 a 1161 vértices). Además, como es habitual en el SMCP, los grafos son poco densos.

En primer lugar, se han realizado una serie de experimentos para analizar el comportamiento de las propuestas algorítmicas descritas en este trabajo y configurar sus parámetros. Posteriormente, se ha comparado el rendimiento de los métodos propuestos con los mejores algoritmos disponibles en el estado del arte para cada uno de los problemas estudiados. Para

Método	$\overline{\text{F.O.}}$	$\overline{\text{Dev. (\%)}}$	# Best (124)	$\overline{\text{CPUt (s)}}$
GRASP-VND	**15,0611**	**<0,01%**	**116**	**28,95**
LNS [104]	15,0374	0,44%	52	36,33

Tabla B.3 Comparación de los resultados obtenidos con el método GRASP-VND propuesto y el método LNS [104] del estado del arte para el problema MQ.

garantizar comparaciones justas, todos los resultados en cada experimento se han obtenido ejecutando los algoritmos en el mismo entorno.

En la Tabla B.3, se muestran los resultados obtenidos en la comparación del método GRASP-VND con el estado del arte para el problema MQ. Para cada método, se muestra el valor medio de la función objetivo MQ ($\overline{\text{F.O.}}$), la desviación media con respecto a la calidad de la mejor solución encontrada por cualquiera de los dos métodos ($\overline{\text{Dev. (\%)}}$), el número de instancias para las que cada método ha encontrado una solución igual o mejor que el otro método (# Best) y el tiempo consumido de media ($\overline{\text{CPUt (s)}}$). Como puede verse, GRASP-VND obtiene mejores soluciones que el estado del arte en menos tiempo, con una desviación media menor al 0,01% y encontrando soluciones iguales o mejores que las del estado del arte en 116 de las 124 instancias. Además, según los resultados del test de Wilcoxon, la diferencia de los resultados es estadísticamente significativa con $p < 0.01$.

En la Tabla B.4, se muestran los resultados obtenidos en la comparación del método GVNS con el estado del arte (HGA [105]) para el problema FCB. Para cada método, se muestra el valor medio de la función objetivo ($\overline{\text{F.O.}}$), la desviación media con respecto a la calidad de la mejor solución encontrada por cualquiera de los dos métodos ($\overline{\text{Dev. (\%)}}$), el número de instancias para las que cada método ha encontrado una solución igual o mejor que el otro método (# Best) y el tiempo consumido de media ($\overline{\text{CPUt (s)}}$). Como puede verse, GVNS obtiene mejores soluciones que el estado del arte en menos tiempo. Además, según los resultados del test de Wilcoxon, la diferencia de los resultados es estadísticamente significativa con $p < 0.001$.

Finalmente, en la Tabla B.5 y en la Tabla B.6, se muestran los resultados obtenidos en la comparación del método MO-GVNS con el estado del arte para los problemas MCA y ECA, respectivamente. Para cada método, se muestra la media del tiempo consumido ($\overline{\text{CPUt (s)}}$), el tamaño de los frentes de Pareto obtenidos ($\overline{\text{Tamaño FP}}$), el hipervolumen

Método	$\overline{\text{F.O.}}$	$\overline{\text{Dev. (\%)}}$	# Best (124)	$\overline{\text{CPUt (s)}}$
GVNS	**0,5821**	**0,00%**	**124**	**136,51**
HGA [105]	0,7340	32,96%	12	27.894,91

Tabla B.4 Comparación de los resultados obtenidos con el método GVNS y el método HGA del estado del arte [105] para el problema FCB.

Método	$\overline{\text{CPUt (s)}}$	$\overline{\text{Tamaño FP}}$	$\overline{\text{HV}}$	$\overline{\text{C}}$	$\overline{\text{IGD+}}$	$\overline{\text{Spread}}$
MO-GVNS	**311,18**	507,60	**0,2213**	**0,0264**	**0,0443**	**0,5175**
MOEA/D [147]	336,43	300,00	0,0982	0,5254	0,2184	0,6844
NSGA-III [31]	596,41	479,56	0,0937	0,2719	0,2587	0,5891
PESA2 [27]	643,16	99,63	0,0604	0,7335	0,3209	0,6994
TA-ABC [8]	1037,31	97,70	0,0277	0,2145	0,3991	0,8026

Tabla B.5 Comparación del método MO-GVNS con los algoritmos de referencia en el estado del arte para el problema MCA.

($\overline{\text{HV}}$), el porcentaje de puntos eficientes dominados ($\overline{\text{C}}$), la distancia al frente de referencia ($\overline{\text{IGD+}}$) y la diversidad de los puntos eficientes ($\overline{\text{Spread}}$). Para obtener los indicadores de calidad mencionados, se ha utilizado un frente aproximado de referencia obtenido mediante la unión de todos los frentes generados por los distintos algoritmos de la comparación. Como puede verse, MO-GVNS obtiene mejores soluciones que el estado del arte en menos tiempo. Además, los frentes de Pareto contienen un mayor número de soluciones.

Método	$\overline{\text{CPUt (s)}}$	$\overline{\text{Tamaño FP}}$	$\overline{\text{HV}}$	$\overline{\text{C}}$	$\overline{\text{IGD+}}$	$\overline{\text{Spread}}$
MO-GVNS	**311,16**	520,64	**0,1939**	**0,0122**	**0,0180**	**0,5614**
MOEA/D [147]	345,06	300,00	0,0724	0,8010	0,3312	0,6032
NSGA-III [31]	745,81	209,56	0,0889	0,5918	0,3690	0,6777
PESA2 [27]	408,69	89,59	0,0443	0,7386	0,4777	0,7599
TA-ABC [8]	914,74	30,66	0,0284	0,3051	0,5132	0,8932

Tabla B.6 Comparación del método MO-GVNS con los algoritmos de referencia en el estado del arte para el problema ECA.

B.8 Conclusiones y trabajos futuros

En esta tesis doctoral, se han estudiado cuatro problemas distintos del SMCP: MQ, FCB, MCA y ECA. Los dos primeros problemas son monoobjetivo, mientras que los dos últimos presentan un enfoque multiobjetivo. Para abordar estos problemas, se han propuesto tres algoritmos basados en metaheurísticas trayectoriales: un método basado en GRASP-VND, un método basado en GVNS y un método basado en MO-GVNS. Los algoritmos propuestos incluyen un procedimiento constructivo semivoraz, un procedimiento constructivo voraz para generar conjuntos de soluciones en las variantes multiobjetivo, distintos mecanismos de perturbación de las soluciones y seis estructuras de vecindad distintas.

Una contribución importante de esta tesis doctoral es el estudio y la clasificación de las estructuras de vecindad para los problemas del SMCP. Tras analizar distintas vecindades, nos dimos cuenta de que estas se pueden clasificar en tres categorías: (1) vecindades cuyas operaciones no están diseñadas para alterar el número de módulos de la solución, (2) vecindades cuyas operaciones están diseñadas para aumentar el número de módulos en la solución y (3) vecindades cuyas operaciones están diseñadas para reducir el número de módulos en la solución. La exploración de al menos una vecindad de cada categoría dota a los métodos propuestos de flexibilidad para aumentar, disminuir o mantener el número de módulos en una solución dada. De las seis estructuras de vecindad propuestas, dos pertenecen a la primera categoría, dos a la segunda y dos a la tercera.

La segunda estrategia avanzada consiste en una evaluación eficiente de las funciones objetivo estudiadas. Debido a la naturaleza de las metaheurísticas trayectoriales, basadas en la mejora gradual de las soluciones mediante pequeñas modificaciones, la factorización de las funciones objetivo permite realizar evaluaciones parciales y acelerar los métodos propuestos.

La tercera estrategia propuesta consiste en una identificación de regiones prometedoras del espacio de búsqueda. En un proceso de búsqueda, existen muchos movimientos que no mejoran la solución. En problemas en los que el espacio de búsqueda es muy grande, es importante identificar regiones prometedoras para acelerar el proceso. De esta manera, se reduce el tamaño de las estructuras de vecindad exploradas y se reduce el esfuerzo computacional de los algoritmos propuestos.

La cuarta estrategia propuesta consiste en un análisis de la contribución de las funciones guía utilizadas durante el proceso de búsqueda. Esta reducción permite disminuir el tiempo computacional empleado por el método MO-GVNS propuesto. No obstante, la reducción de las funciones guía conlleva una reducción de la calidad de las soluciones obtenidas, por lo que es necesario encontrar un equilibrio entre coste computacional y calidad de las soluciones. Es importante destacar que lo que se busca es reducir el número de funciones guía, no el número de funciones objetivo utilizadas para evaluar las soluciones.

Finalmente, los métodos propuestos se han comparado favorablemente con los mejores algoritmos disponibles en el estado del arte, obteniendo mejores soluciones con un menor coste computacional. Por lo tanto, se puede afirmar que los resultados obtenidos apoyan la hipótesis enunciada en esta tesis doctoral: que las metaheurísticas trayectoriales pueden obtener mejores resultados que los métodos poblacionales para la optimización de la estructura de sistemas *software* en términos de modularidad. Además, junto a la hipótesis de partida, se formularon diversos objetivos parciales que se han cumplido a lo largo de esta tesis doctoral. En la Sección B.5, se puede encontrar una revisión del estado del arte del SMCP. La definición de los problemas y la representación de las soluciones se pueden encontrar en la Sección B.4. El conjunto de instancias obtenido se describe en la Sección B.7 y, más detalladamente, en el apéndice A. Los algoritmos de referencia del estado del arte se han implementado y los resultados obtenidos se incluyen en la comparación descrita en la Sección B.7. Los algoritmos propuestos se describen en la Sección B.6. Por brevedad, los resultados de estos experimentos no se han recogido en este apéndice, pero se pueden encontrar en la Sección 5.2 de esta tesis doctoral. La comparación con los mejores algoritmos disponibles para cada problema estudiado se describe en la Sección B.7. Finalmente, las contribuciones realizadas durante el desarrollo de esta tesis doctoral se recogen en la Sección B.9. En resumen, se puede considerar que se han cumplido todos los objetivos parciales enunciados en esta tesis doctoral. Además, los resultados obtenidos parecen ser de interés para la comunidad científica, ya que han sido publicados en diferences foros científicos.

En trabajos futuros, sería interesante explorar la hibridación de metaheurísticas trayectoriales con metaheurísticas poblaciones. Más aun, aunque la naturaleza del problema hace

que no sea posible utilizar algoritmos exactos para instancias de gran tamaño, sería interesante explorar la combinación de métodos exactos y metaheurísticas en algoritmos de tipo *matheuristic*.

En relación al estudio de los diferentes algoritmos propuestos para el SMCP, una línea de trabajo futuro consiste en comparar el rendimiento de estos métodos en distintas variantes del problema. Este análisis podría ayudar a identificar componentes comunes y diferencias entre las distintas métricas, así como posibles transferencias de estrategias de búsqueda entre distintas variantes del problema.

Otra línea de trabajo futuro es la integración de los métodos propuestos en el SDLC de proyectos *software*. En particular, el bajo coste computacional de los métodos analizados permitiría su inclusión en entornos de desarrollo integrados o plataformas de integración continua. En este sentido, los desarrolladores se podrían beneficiar de sugerencias para reestructurar los proyectos en tiempo real. No obstante, para integrar los métodos propuestos en el SDLC de proyectos *software*, sería beneficioso mantener un equilibrio entre la mejora de la calidad del código y la magnitud de los cambios. Este balance evitaría que los desarrolladores perdieran la familiaridad que tienen con el sistema en cuestión. En este sentido, se podría explorar la introducción en un enfoque multiobjetivo de la minimización de los cambios en el código.

B.9 Contribuciones

Durante el desarrollo de esta tesis doctoral, se han realizado diversas publicaciones y presentaciones en foros científicos. En la Figura B.8, se presenta una cronología que contiene los eventos más relevantes con relación a esta tesis doctoral. Cada evento se ha destacado con un color diferente dependiendo de la siguiente clasificación: los artículos de investigación publicados en revistas clasificadas en el ránking JCR[1] (del inglés *Journal Citation Reports*) o en el ránking SJR[2] (del inglés *Scientific Journal Rankings*) se destacan en color rojo (■); presentaciones realizadas en congresos y talleres se destacan en color verde (■);

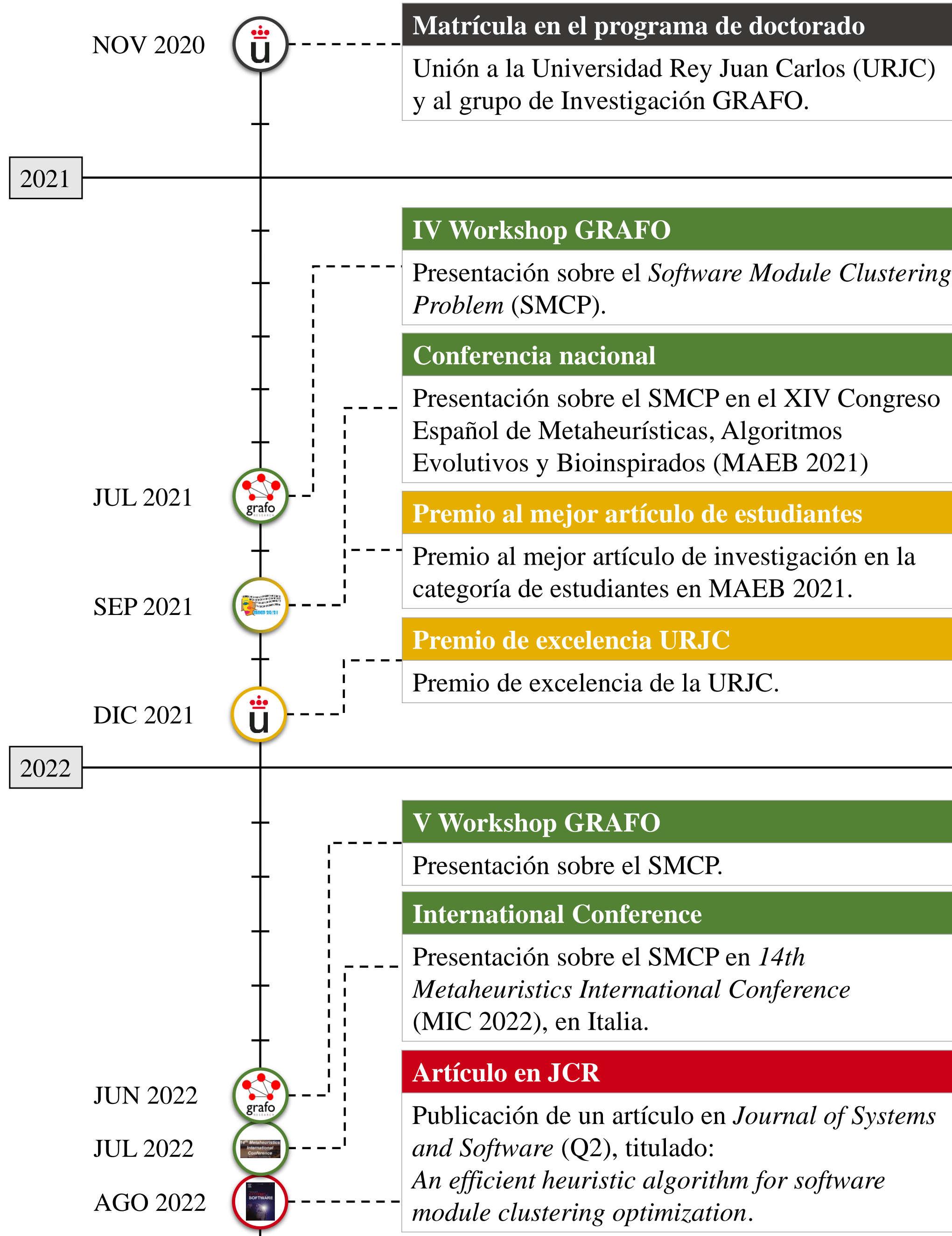
[1] `https://jcr.clarivate.com/jcr/home`
[2] `https://www.scimagojr.com/journalrank.php`

los premios y reconocimientos se destacan en color amarillo (); las estancias de investigación se destacan en color morado (); por último, los eventos que marcan el inicio o el final del desarrollo de esta tesis doctoral se destacan en color gris oscuro ().

Como se puede observar, el desarrollo de esta tesis doctoral comenzó en noviembre de 2020, cuando el autor se unió al programa de doctorado de la Universidad Rey Juan Carlos (URJC). Al mismo tiempo, el autor entraba a formar parte del grupo de investigación GRAFO (del inglés *Group for Research in Algorithms For Optimization*). En 2021, algunos hallazgos preliminares fueron presentados en el IV Workshop GRAFO y en el XIV Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB) [142], que se celebró en Málaga, España. El trabajo presentado en esta conferencia nacional fue reconocido con el premio al mejor artículo de investigación de la conferencia en la categoría de estudiantes. Como resultado, el autor de esta tesis doctoral recibió el premio de excelencia de la URJC más tarde ese mismo año.

En 2022, parte de la investigación de esta tesis doctoral fue presentada en el V Workshop GRAFO y en la decimocuarta edición del congreso internacional MIC (del inglés *Metaheuristics International Conference*), celebrada en Siracusa, Italia. Los resultados presentados en esta conferencia internacional se publicaron en LNCS (del inglés *Lecture Notes in Computer Science*) [144], una revista clasificada en el tercer cuartil (Q3) del ránking SJR. Además, en ese mismo año, se publicó un artículo de investigación en la revista *Journal of Systems and Software*, clasificada en el segundo cuartil (Q2) del ránking JCR, con el título: "*An efficient heuristic algorithm for software module clustering optimization*" [141]. A finales de 2022, se realizó una presentación en la tercera edición de la Escuela de Invierno organizada por la Red HEUR, en Burgos, España.

En 2023, se publicó un segundo artículo en LNCS [143]. Además, se realizó una estancia internacional en la Université d'Angers, en Angers, Francia. Durante esta estancia de tres meses de duración, el autor de esta tesis doctoral tuvo el honor de colaborar con el profesor Jin-Kao Hao. Como resultado de esta colaboración, se realizó un artículo de investigación que se encuentra actualmente en proceso de revisión en una revista clasificada en el ránking JCR. Finalmente, durante el año 2023, se realizaron tres presentaciones. La primera se realizó en el I Workshop OptiMAD, celebrado en Madrid, España; la segunda se realizó en el VI Workshop GRAFO; y la tercera se realizó en el XL Congreso



continúa en la siguiente página

Figura B.8 Cronología de los eventos más relevantes relacionados con esta tesis doctoral.

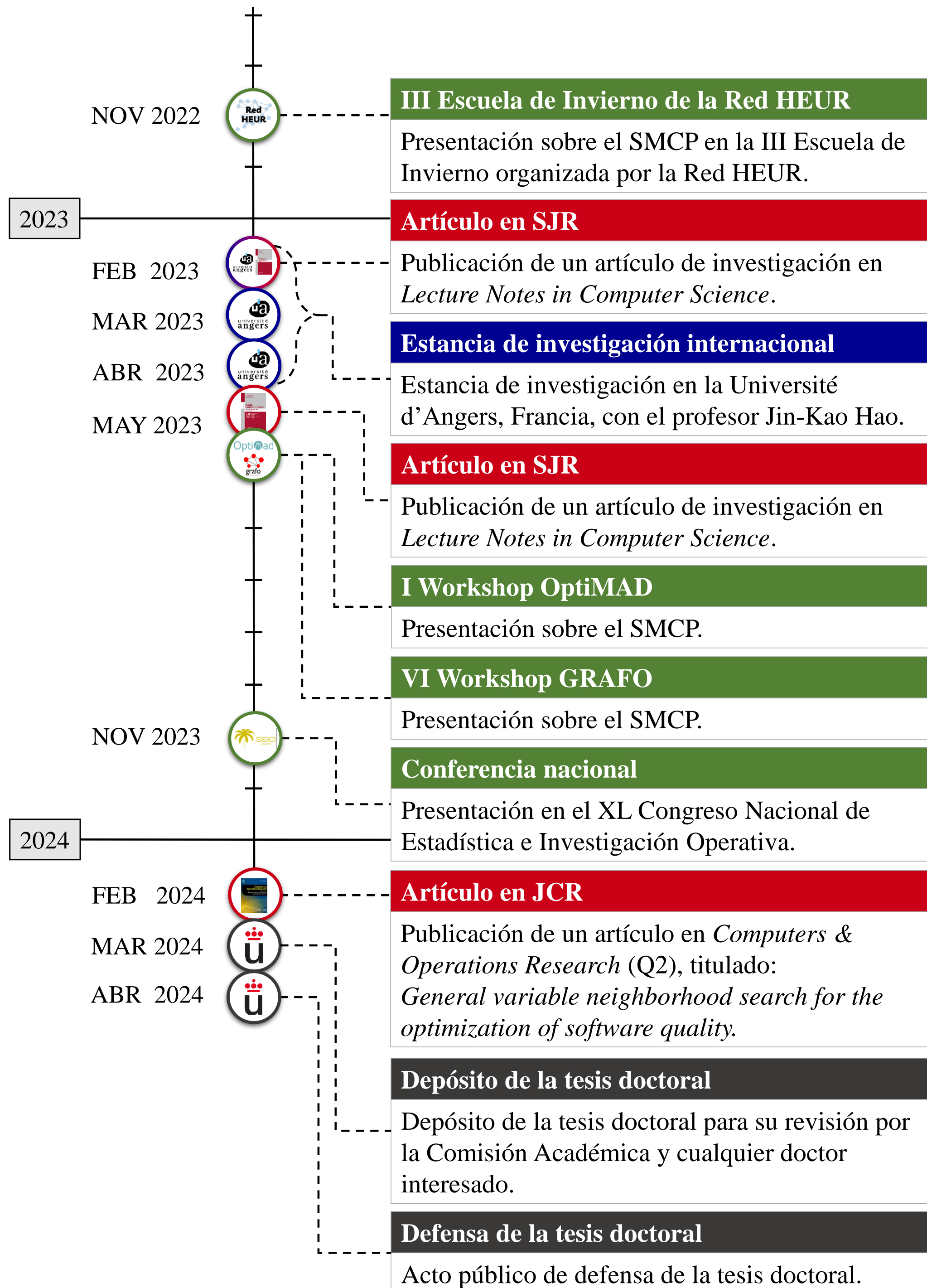


Figura B.8 Cronología de los eventos más relevantes relacionados con esta tesis doctoral.

Nacional de Estadística e Investigación Operativa, en Elche, España.

Durante el último año de desarrollo de esta tesis doctoral, 2024, se publicó un artículo de investigación en la revista *Computers & Operations Research*, clasificada en el segundo cuartil (Q2) del ránking JCR, con el título: "*General Variable Neighborhood Search for the optimization of software quality*". Además, durante el primer semestre de este año, se depositó la memoria de esta tesis doctoral y se defendió la misma.

A modo de resumen, se han realizado las siguientes contribuciones durante el desarrollo de esta tesis doctoral:

- Artículos publicados en revistas indexadas en el JCR:

  1. J. Yuste, A. Duarte y E. G. Pardo. An efficient heuristic algorithm for software module clustering optimization. *Journal of Systems and Software*, 190: 111349, 2022.
  2. J. Yuste, E. G. Pardo y A. Duarte. General Variable Neighborhood Search for the optimization of software quality. *Computers & Operations Research*, 106584, 2024.
  3. J. Yuste, E. G. Pardo, A. Duarte y J. Hao. Multi-Objective General Variable Neighborhood Search for Software Maintainability Optimization. *Engineering Applications of Artificial Intelligence*, EN REVISIÓN.

- Artículos publicados en revistas indexadas en el SJR:

  1. J. Yuste, E. G. Pardo y A. Duarte. Variable neighborhood descent for software quality optimization. En *Metaheuristics International Conference*, páginas 531-536. Springer, 2022.
  2. J. Yuste, E. G. Pardo y A. Duarte. Multi-objective variable neighborhood search for improving software modularity. En *International Conference on Variable Neighborhood Search*, páginas 58–68. Springer, 2022.

- Presentaciones en conferencias internacionales:

  1. J. Yuste, E. G. Pardo y A. Duarte. Variable neighborhood descent for software quality optimization. En *14th Metaheuristics International Conference*, Siracusa, Italia. 11-14 de julio, 2022.

- Presentaciones en conferencias nacionales:

  1. J. Yuste, E. G. Pardo y A. Duarte. Heurísticas para la mejora de la mantenibilidad de proyectos software. En *XIX Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA 20/21)*, Málaga, España, páginas 581–586. 22-24 de septiembre, 2021.
  2. J. Yuste, E. G. Pardo, A. Duarte y J. Hao. Optimización multiobjetivo en problemas de calidad de software. En *XL Congreso Nacional de Estadística e Investigación Operativa*, Elche, España. 7-10 de noviembre, 2023.

- Presentaciones en talleres:

  1. J. Yuste, E. G. Pardo y A. Duarte. Software Module Clustering Problem. En *IV Workshop GRAFO*, Móstoles, España. 14-15 de julio, 2021.
  2. J. Yuste, E. G. Pardo y A. Duarte. Software Module Clustering Problem. En *V Workshop GRAFO*, Móstoles, España. 2-3 de junio, 2022.
  3. J. Yuste, E. G. Pardo, A. Duarte y J. Hao. Optimización de la calidad de los sistemas software: una aproximación multiobjetivo. En *I OptiMad*, Madrid, España. 25 de mayo, 2023.
  4. J. Yuste, E. G. Pardo y A. Duarte. Software Module Clustering Problem. En *VI Workshop GRAFO*, Móstoles, España. 30-31 de mayo, 2023.

- Estancia internacional de investigación:

  1. Realizada en la Université d'Angers, en Angers, Francia, bajo la supervisión del profesor Jin-Kao Hao. Un artículo se realizó durante esta estancia en colaboración con el profesor Jin-Kao Hao. A fecha de escritura de esta memoria, el

artículo mencionado se encuentra en proceso de revisión en una revista clasificada en el ránking JCR.

Bibliography

[1] Chaos Report 2015. Technical report, The Standish Group, 2015. URL: https://standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf. Accessed 20 Dec. 2023.

[2] “Research”. In *Merriam-Webster.com Dictionary*. Merriam-Webster, 2023. URL: https://www.merriam-webster.com/dictionary/research. Accessed 18 Sep. 2023.

[3] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui. Automatic package coupling and cycle minimization. In *2009 16th Working Conference on Reverse Engineering*, pages 103–112. IEEE, 2009.

[4] J. Al Dallal. Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology*, 55(11):2028–2048, 2013.

[5] A. Albee, S. Battel, R. Brace, G. Burdick, J. Casani, J. Lavell, C. Leising, D. MacPherson, P. Burr, and D. Dipprey. Report on the loss of the mars polar lander and deep space 2 missions. *NTRS - NASA Technical Reports Server*, 2000.

[6] S. Almugrin, W. Albattah, and A. Melton. Using indirect coupling metrics to predict package maintainability and testability. *Journal of systems and software*, 121:298–310, 2016.

[7] Amarjeet and J. K. Chhabra. Harmony search based remodularization for object-oriented software systems. *Computer Languages, Systems & Structures*, 47:153–169, 2017.

[8] Amarjeet and J. K. Chhabra. TA-ABC: Two-Archive Artificial Bee Colony for Multi-objective Software Module Clustering Problem. *Journal of Intelligent Systems*, 27(4):619–641, 2018.

[9] B. Arasteh, A. Fatolahzadeh, and F. Kiani. Savalan: Multi objective and homogeneous method for software modules clustering. *Journal of Software: Evolution and Process*, 34(1):e2408, 2022.

[10] P. Arcaini, T. Yue, and E. M. Fredericks. *Search-Based Software Engineering: 15th International Symposium, SSBSE 2023, San Francisco, CA, USA, December 8, 2023, Proceedings*, volume 14415. Springer Cham, 2023.

[11] 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. Proceedings 3*, pages 33–47. Springer, 2011.

[12] C. S. Atole and K. Kale. Assessment of package cohesion and coupling principles for predicting the quality of object oriented design. In *2006 1st International Conference on Digital Information Management*, pages 1–5. IEEE, 2006.

[13] T. Bakota, P. Hegedus, G. Ladanyi, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A cost model based on software maintainability. In *IEEE International Conference on Software Maintenance, ICSM*, pages 316–325, 2012.

[14] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. Resende, and W. R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9–32, sep 1995.

[15] M. O. Barros. An analysis of the effects of composite objectives in multiobjective software module clustering. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 1205–1212, 2012.

[16] M. O. Barros, d. Fábio, and H. Guilherme. Learning from optimization: A case study with Apache Ant. *Information and Software Technology*, 57:684–704, 2015.

[17] G. Bavota, M. Di Penta, and R. Oliveto. Search based software maintenance: Methods and tools. *Evolving software systems*, pages 103–137, 2014.

[18] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2007.

[19] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3):245–273, 2000.

[20] S. Cavero, E. G. Pardo, and A. Duarte. A general variable neighborhood search for the cyclic antibandwidth problem. *Computational Optimization and Applications*, pages 1–31, 2022.

[21] S. Cavero, E. G. Pardo, A. Duarte, and E. Rodriguez-Tello. A variable neighborhood search approach for cyclic bandwidth sum problem. *Knowledge-Based Systems*, 246:108680, 2022.

[22] C. Chen, R. Alfayez, K. Srisopha, B. Boehm, and L. Shi. Why is it important to measure maintainability and what are the best ways to do it? In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 377–378. IEEE, 2017.

[23] J. C. Chen and S. J. Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992, 2009.

[24] J. K. Chhabra. Many-objective artificial bee colony algorithm for large-scale software module clustering problem. *Soft Computing*, 22(19):6341–6361, 2018.

[25] T. E. Colanzi, W. K. Assunção, S. R. Vergilio, P. R. Farah, and G. Guizzo. The symposium on search-based software eengineering: Past, present and future. *Information and Software Technology*, 127:106372, 2020.

[26] T. E. Colanzi, S. R. Vergilio, I. M. Gimenes, and W. N. Oizumi. A search-based approach for software product line design. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 237–241, 2014.

[27] D. W. Corne, N. R. Jerram, J. D. Knowles, and M. J. Oates. PESA-II: Region-based selection in evolutionary multiobjective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 283–290, 2001.

[28] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[29] G. B. Dantzig. Linear programming. *Operations research*, 50(1):42–47, 2002.

[30] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Parallel Problem Solving from Nature PPSN VI: 6th International Conference Paris, France, September 18–20, 2000 Proceedings 6*, pages 849–858. Springer, 2000.

[31] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2013.

[32] K. Deep and M. Thakur. A new crossover operator for real coded genetic algorithms. *Applied mathematics and computation*, 188(1):895–911, 2007.

[33] K. Deep and M. Thakur. A new mutation operator for real coded genetic algorithms. *Applied mathematics and Computation*, 193(1):211–230, 2007.

[34] S. Dekleva. Delphi study of software maintenance problems. In *Proceedings Conference on Software Maintenance 1992*, pages 10–11. IEEE Computer Society, 1992.

[35] M. Di Penta. SBSE meets software maintenance: Achievements and open problems. In *International Symposium on Search Based Software Engineering*, pages 27–28. Springer, 2012.

[36] M. Di Penta, M. Harman, and G. Antoniol. The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. *Software: Practice and Experience*, 41(5):495–519, 2011.

[37] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *STEP'99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, pages 73–81. IEEE, 1999.

[38] A. Duarte, J. J. Pantrigo, and M. Gallego. Metaheurísticas. *Madrid: Dykinson*, 2007.

[39] A. Duarte, J. J. Pantrigo, E. G. Pardo, and N. Mladenovic. Multi-objective variable neighborhood search: an application to combinatorial optimization problems. *Journal of Global Optimization*, 63(3):515–536, 2015.

[40] F. Y. Edgeworth. *Mathematical psychics: An essay on the application of mathematics to the moral sciences*. Number 10. C. Kegan Paul and Co, London, 1881.

[41] M. Ehrgott. *Multicriteria Optimization*, volume 491. Springer Science & Business Media, 2005.

[42] S. Ergasheva and A. Kruglov. Software Development Life Cycle early phases and quality metrics: A Systematic Literature Review. In *Journal of Physics: Conference Series*, volume 1694, page 012007. IOP Publishing, 2020.

[43] T. A. Feo and M. G. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.

[44] T. A. Feo and M. G. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[45] F. Ferrucci, M. Harman, and F. Sarro. Search-based software project management. *Software project management in a changing world*, pages 373–399, 2014.

[46] C. A. Floudas and P. M. Pardalos. *Encyclopedia of optimization*. Springer Science & Business Media, 2008.

[47] F. Formica, T. Fan, and C. Menghi. Search-based software testing driven by automatically generated and manually defined fitness functions. *ACM Transactions on Software Engineering and Methodology*, 2022.

[48] F. García-López, B. Melián-Batista, J. A. Moreno-Pérez, and J. M. Moreno-Vega. The parallel variable neighborhood search for the p-median problem. *Journal of Heuristics*, 8:375–388, 2002.

[49] M. Gendreau, A. Hertz, and G. Laporte. New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, 40(6):1086–1094, 1992.

[50] M. Gendreau and J.-Y. Potvin. *Handbook of Metaheuristics*, volume 2. Springer, 2010.

[51] S. Gil-Borrás, E. G. Pardo, A. Alonso-Ayuso, and A. Duarte. A heuristic approach for the online order batching problem with multiple pickers. *Computers & Industrial Engineering*, 160:107517, 2021.

[52] F. Glover. Tabu search and adaptive memory programming—advances, applications and challenges. *Interfaces in computer science and operations research: Advances in metaheuristics, optimization, and stochastic modeling technologies*, pages 1–75, 1997.

[53] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[54] M. Harman and B. F. Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[55] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012.

[56] J. Huang and J. Liu. A similarity-based modularization quality measure for software module clustering problems. *Information Sciences*, 342:96–110, 2016.

[57] J. Huang, J. Liu, and X. Yao. A multi-agent evolutionary algorithm for software module clustering problems. *Soft Computing*, 21(12):3415–3428, 2017.

[58] W. Humphrey. Introduction to Software Process Improvement. Technical Report CMU/SEI-92-TR-007, Carnegie Mellon University, Software Engineering Institute's Digital Library, 1992. URL: `https://doi.org/10.1184/R1/6574820.v1`. Accessed 20 Dec. 2023.

[59] J. Hwa, S. Yoo, Y.-S. Seo, and D.-H. Bae. Search-based approaches for software module clustering based on multiple relationship factors. *International Journal of Software Engineering and Knowledge Engineering*, 27(07):1033–1062, 2017.

[60] A. Isazadeh, H. Izadkhah, and I. Elgedawy. *Source code modularization: theory and techniques*. Springer, 2017.

[61] ISO/IEC/IEEE 12207:2017 Systems and software engineering — Software life cycle processes. Technical report, International Organization for Standardization, Geneva, CH, 2017.

[62] ISO 24156-1:2014 Graphic notations for concept modelling in terminology work and its relationship with UML. Technical report, International Organization for Standardization, Geneva, CH, 2020.

[63] ISO/IEC/IEEE 24765:2017 Systems and software engineering — Vocabulary. Technical report, International Organization for Standardization, Geneva, CH, 2017.

[64] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Technical report, International Organization for Standardization, Geneva, CH, 2011.

[65] H. Izadkhah, I. Elgedawy, and A. Isazadeh. E-CDGM: an evolutionary call-dependency graph modularization approach for software systems. *Cybernetics and Information Technologies*, 16(3):70–90, 2016.

[66] H. Izadkhah and M. Tajgardan. Information Theoretic Objective Function for Genetic Software Clustering. In *Multidisciplinary Digital Publishing Institute Proceedings*, volume 46, page 18, 2019.

[67] N. S. Jalali, H. Izadkhah, and S. Lotfi. Multi-objective search-based software modularization: structural and non-structural features. *Soft Computing*, 23(21):11141–11165, 2019.

[68] K. Jeet and R. Dhir. Software module clustering using hybrid socio-evolutionary algorithms. *International Journal of Information Engineering and Electronic Business*, 8(4):43, 2016.

[69] B. F. Jones, D. E. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.

[70] B. F. Jones, H.-H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software engineering journal*, 11(5):299–306, 1996.

[71] D. Karaboga. An idea based on honey bee swarm for numerical optimization. Technical report, Technical report-tr06, Erciyes university, engineering faculty, 2005.

[72] D. Karaboga, B. Gorkemli, C. Ozturk, and N. Karaboga. A comprehensive survey: artificial bee colony (ABC) algorithm and applications. *Artificial intelligence review*, 42:21–57, 2014.

[73] M. Kargar, A. Isazadeh, and H. Izadkhah. Semantic-based software clustering using hill climbing. In *2017 International Symposium on Computer Science and Software Engineering Conference (CSSE)*, pages 55–60. IEEE, 2017.

[74] M. Khari and P. Kumar. An extensive evaluation of search-based software testing: a review. *Soft Computing*, 23:1933–1946, 2019.

[75] V. Köhler, M. Fampa, and O. Araújo. Mixed-integer linear programming formulations for the software clustering problem. *Computational Optimization and Applications*, 55(1):113–135, 2013.

[76] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4:87–112, 1994.

[77] H. Krasner. The Cost of Poor Software Quality in the US: A 2020 Report. Technical report, Consortium for Information & Software Quality, 2020. URL: `https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/`. Accessed 12 Sep. 2023.

[78] A. C. Kumari and K. Srinivas. Hyper-heuristic approach for multi-objective software module clustering. *Journal of Systems and Software*, 117:384–401, 2016.

[79] G. Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, 1992.

[80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[81] M. M. Lehman. Laws of software evolution revisited. In *European workshop on software process technology*, pages 108–124. Springer, 1996.

[82] B. P. Lientz and E. B. Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[83] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed. A systematic mapping study of search-based software engineering for software product lines. *Information and software technology*, 61:33–51, 2015.

[84] I. Lozano-Osorio, J. Sanchez-Oro, A. Duarte, and Ó. Cordón. A quick GRASP-based method for influence maximization in social networks. *Journal of Ambient Intelligence and Humanized Computing*, 14(4):3767–3779, 2023.

[85] K. Mahdavi. *A clustering genetic algorithm for software modularisation with a multiple hill climbing approach.* PhD thesis, Brunel University, UK, 2005.

[86] A. S. Mamaghani and M. Hajizadeh. Software modularization using the modified firefly algorithm. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*, pages 321–324. IEEE, 2014.

[87] A. S. Mamaghani and M. R. Meybodi. Clustering of software systems using new hybrid algorithms. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, volume 1, pages 20–25. IEEE, 2009.

[88] 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 IEEE International Conference on Software Maintenance-1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No. 99CB36360)*, pages 50–59. IEEE, 1999.

[89] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *6th International Workshop on Program Comprehension (IWPC'98)*, pages 45–52. IEEE, 1998.

[90] R. Martín-Santamaría, S. Cavero, A. Herrán, A. Duarte, and J. M. Colmenar. A practical methodology for reproducible experimentation: an application to the double-row facility layout problem. *Evolutionary Computation*, pages 1–35, 2022.

[91] R. Martí, P. M. Pardalos, and M. G. Resende. *Handbook of heuristics*. Springer Publishing Company, Incorporated, 2018.

[92] M. McFall-Johnsen. Catastrophic software errors doomed Boeing's airplanes and nearly destroyed its NASA spaceship. Experts blame the leadership's 'lack of engineering culture'. Technical report, Business Insider, 2020. URL: https://www.businessinsider.com/boeing-software-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2. Accessed 12 Sep. 2023.

[93] P. McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[94] B. Mitchell, M. Traverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 181–190. IEEE, 2001.

[95] B. S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel University, USA, 2002. AAI3039424.

[96] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382, 2002.

[97] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[98] B. S. Mitchell and S. Mancoridis. On the evaluation of the Bunch search-based software modularization algorithm. *Soft Computing*, 12(1):77–93, 2008.

[99] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[100] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[101] N. Mladenović, F. Plastria, and D. Urošević. Reformulation descent applied to circle packing problems. *Computers & Operations Research*, 32(9):2419–2434, 2005.

[102] M. Mohan and D. Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1):1–52, 2018.

[103] A.-J. Molnar and S. Motogna. A study of maintainability in evolving open-source software. In *Evaluation of Novel Approaches to Software Engineering: 15th International Conference, ENASE 2020, Prague, Czech Republic, May 5–6, 2020, Revised Selected Papers 15*, pages 261–282. Springer, 2021.

[104] M. C. Monçores, A. C. F. Alvim, and M. O. Barros. Large neighborhood search applied to the software module clustering problem. *Computers & Operations Research*, 91:92–111, 2018.

[105] L. Mu, V. Sugumaran, and F. Wang. A hybrid genetic algorithm for software architecture re-modularization. *Information Systems Frontiers*, 22(5):1133–1161, 2020.

[106] OECD. *Frascati Manual 2015*. 2015. URL: `https://www.oecd-ilibrary.org/content/publication/9789264239012-en`. Accessed 20 Dec. 2023.

[107] J. J. Pantrigo, R. Martí, A. Duarte, and E. G. Pardo. Scatter search for the cutwidth minimization problem. *Annals of Operations Research*, 199:285–304, 2012.

[108] E. G. Pardo, A. García-Sánchez, M. Sevaux, and A. Duarte. Basic variable neighborhood search for the minimum sitting arrangement problem. *Journal of Heuristics*, 26:249–268, 2020.

[109] E. G. Pardo, S. Gil-Borrás, A. Alonso-Ayuso, and A. Duarte. Order batching problems: taxonomy and literature review. *European Journal of Operational Research*, 2023.

[110] E. G. Pardo, N. Mladenović, J. J. Pantrigo, and A. Duarte. Variable formulation search for the cutwidth minimization problem. *Applied Soft Computing*, 13(5):2242–2252, 2013.

[111] V. Pareto. *Cours d'économie politique*, volume 1. Librairie Droz, 1964.

[112] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software testing, verification and reliability*, 9(4):263–282, 1999.

[113] R. Pellerin, N. Perrier, and F. Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 280(2):395–416, 2020.

[114] S. Perez-Pelo, J. Sanchez-Oro, A. Gonzalez-Pardo, and A. Duarte. A fast variable neighborhood search approach for multi-objective community detection. *Applied Soft Computing*, 112:107838, 2021.

[115] V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, 2013.

[116] A. F. Pinto, A. C. de Faria Alvim, and M. O. Barros. ILS for the Software Module Clustering Problem. *XLVI Simpósio Brasileiro de Pesquisa Operacional. Salvador:[sn]*, pages 1972–1983, 2014.

[117] A. M. Pitangueira, R. S. P. Maciel, and M. O. Barros. Software requirements selection and prioritization using sbse approaches: A systematic review and mapping of the literature. *Journal of Systems and Software*, 103:267–280, 2015.

[118] G. Polya. *How to solve it; a new aspect of mathematical method.* Princeton University Press, 1945.

[119] B. Pourasghar, H. Izadkhah, A. Isazadeh, and S. Lotfi. A graph-based clustering algorithm for software systems modularization. *Information and Software Technology*, 133:106469, 2021.

[120] K. Praditwong. Solving software module clustering problem by evolutionary algorithms. In *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 154–159. IEEE, 2011.

[121] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[122] A. Prajapati. Software module clustering using grid-based large-scale many-objective particle swarm optimization. *Soft Computing*, pages 1–22, 2022.

[123] A. Prajapati and J. K. Chhabra. A particle swarm optimization-based heuristic for software module clustering problem. *Arabian Journal for Science and Engineering*, 43(12):7083–7094, 2018.

[124] O. Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[125] A. Ramírez, P. Delgado-Pérez, J. Ferrer, J. R. Romero, I. Medina-Bulo, and F. Chicano. A systematic literature review of the SBSE research community in Spain. *Progress in Artificial Intelligence*, 9:113–128, 2020.

[126] A. Ramírez, J. A. Parejo, J. R. Romero, S. Segura, and A. Ruiz-Cortés. Evolutionary composition of QoS-aware web services: a many-objective perspective. *Expert Systems with Applications*, 72:357–370, 2017.

[127] A. Ramírez, J. R. Romero, and S. Ventura. A comparative study of many-objective evolutionary algorithms for the discovery of software architectures. *Empirical Software Engineering*, 21(6):2546–2600, 2016.

[128] A. Ramirez, J. R. Romero, and S. Ventura. Interactive multi-objective evolutionary optimization of software architectures. *Information Sciences*, 463:92–109, 2018.

[129] A. Ramirez, J. R. Romero, and S. Ventura. A survey of many-objective optimisation in search-based software engineering. *Journal of Systems and Software*, 149:382–395, 2019.

[130] A. V. Rezende, L. Silva, A. Britto, and R. Amaral. Software project scheduling problem in the context of search-based software engineering: A systematic review. *Journal of Systems and Software*, 155:43–56, 2019.

[131] J. R. Romero, I. Medina-Bulo, and F. Chicano, editors. *Optimising the Software Development Process with Artificial Intelligence*. Natural Computing Series. Springer Nature Singapore, 2023.

[132] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.

[133] M. Tajgardan, H. Izadkhah, and S. Lotfi. Software systems clustering using estimation of distribution approach. *Journal of Applied Computer Science Methods*, 8, 2016.

[134] E.-G. Talbi. *Metaheuristics - From Design to Implementation*. John Wiley & Sons, Inc., 2009.

[135] The European Space Agency. N° 33–1996: Ariane 501 - Presentation of Inquiry Board report. Technical report, 1996. URL: https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report. Accessed 12 Sep. 2023.

[136] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 73–81, 1998.

[137] S. Vathsavayi, O. Räihä, K. Koskimies, et al. Tool support for software architecture design with genetic algorithms. In *2010 Fifth International Conference on Software Engineering Advances*, pages 359–366. IEEE, 2010.

[138] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*, 1996.

[139] W. Y. Wong, S. W. Yu, and C. W. Too. A systematic approach to software quality assurance: the relationship of project activities within project life cycle and system development life cycle. In *2018 IEEE Conference on Systems, Process and Control (ICSPC)*, pages 123–128. IEEE, 2018.

[140] Y. Yuan, Y.-S. Ong, A. Gupta, and H. Xu. Objective reduction in many-objective optimization: evolutionary multiobjective approaches and comprehensive analysis. *IEEE Transactions on Evolutionary Computation*, 22(2):189–210, 2017.

[141] J. Yuste, A. Duarte, and E. G. Pardo. An efficient heuristic algorithm for software module clustering optimization. *Journal of Systems and Software*, 190:111349, 2022.

[142] J. Yuste, E. G. Pardo, and A. Duarte. Heurísticas para la mejora de la mantenibilidad de proyectos software. In *XIX Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA 20/21)*, pages 581–586, 2021.

[143] J. Yuste, E. G. Pardo, and A. Duarte. Multi-objective variable neighborhood search for improving software modularity. In *International Conference on Variable Neighborhood Search*, pages 58–68. Springer, 2022.

[144] J. Yuste, E. G. Pardo, and A. Duarte. Variable neighborhood descent for software quality optimization. In *Metaheuristics International Conference*, pages 531–536. Springer, 2022.

[145] J. Yuste, E. G. Pardo, and A. Duarte. General variable neighborhood search for the optimization of software quality. *Computers & Operations Research*, page 106584, 2024.

[146] S. H. Zanakis and J. R. Evans. Heuristic "optimization": Why, when, and how to use it. *Interfaces*, 11(5):84–91, 1981.

[147] Q. Zhang and H. Li. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

[148] E. Zitzler and S. Künzli. Indicator-based selection in multiobjective search. In *International conference on parallel problem solving from nature*, pages 832–842. Springer, 2004.

Glossary

A

Advanced strategy, 86, 105, 113, 132
  Analysis of guiding functions, 95, 121, 133
  Categorization of neighborhoods, 75, 76, 79, 82, 132
  Incremental evaluation, 86, 87, 89, 119, 125, 132
  Promising regions in the search space, 90, 119, 125, 133
Approximate algorithms, 11, 34, 90
Artificial Bee Colony, 37, 38, 42, 43, 127, 171

B

Basin of attraction, 12

C

Cluster Factor, 73, 86, 87
Cohesion, 20, 23, 24, 26–29, 32, 33, 41, 87, 89, 92, 95, 116, 161, 163–165
Complexity class, 34, 134
Constraints, 8
Constructive procedure, 11, 66, 69–72
Coupling, 20, 21, 23, 25–27, 29, 32, 33, 87, 89, 92, 95, 116, 161, 163, 165
Coverage, 115–117, 119–121, 123, 124, 128

D

Decision variables, 8, 9
Destruction operation, 82, 89, 95
Dominance, 10

E

Efficient point, 58, 69–72, 95, 96, 115, 129
Equal-size Cluster Approach, 18, 27, 29, 33, 37, 38, 42, 45, 68, 69, 75, 89, 90, 96, 97, 116, 117, 119, 121, 124, 127, 128, 131, 133, 157, 165, 166
Exact algorithms, 11, 134
Extraction operation, 79, 95

F

Feasible solutions, 8
Function of Complexity Balance, 18, 23–26, 33, 35, 36, 38, 40, 41, 45, 66, 75, 87, 89–92, 95, 126, 131, 133, 157, 161, 162

G

Generalized Spread, 43, 115–117, 119–121, 123, 124
Genetic Algorithm, 34, 35, 37, 40, 41, 169, 171
Global optimum, 12
Graph, 17, 94
Greedy Randomized Adaptive Search Procedure,

34, 36, 45–48, 59, 63, 102, 103, 105, 124–126, 131, 170

H

Heuristics, 11, 12
Hybrid Genetic Algorithm, 34–36, 38, 40, 41, 126–128, 169, 170
Hypervolume, 43, 115–117, 119–121, 123, 124, 128
Hypothesis and objectives, 13

I

IGD, 43
IGD+, 115–117, 119–121, 123, 124, 128
Insert operation, 76, 87, 89, 91–93

L

Large Neighborhood Search, 34, 38–40, 124–126
Lehman's laws of software evolution, 5
Local optimum, 12, 46, 50, 58, 112
Local search, 11, 46, 47

M

Maximizing Cluster Approach, 18, 26–28, 33, 37, 38, 42, 45, 68, 69, 75, 89, 90, 96, 97, 116, 117, 119, 121, 123, 127, 128, 131, 133, 157, 162–165
Merge operation, 84, 95
Metaheuristics, 12, 34
Methodology, 12, 13
Modularity, 6, 7, 32
Modularization Quality, 18, 20, 22–24, 27, 33, 35, 36, 39, 40, 70, 92, 116, 157, 158
  BasicMQ, 20, 22, 23, 33
  Cluster Factor, 22, 159
  Inter-connectivity, 20, 21
  Intra-connectivity, 20, 21
  TurboMQ, 20, 22, 23, 27, 29, 33, 38, 45, 59, 61, 70, 86–89, 91–93, 95, 102, 105, 124, 131, 133, 158, 160, 163–165
Module Dependency Graph, 17–20, 22, 24, 25, 28, 29, 71
Motivation, 4
Multi-Objective optimization, 9, 10

N

Neighborhood, 11, 50, 56, 58, 72, 75, 76, 78, 79, 81, 82, 84, 86, 91, 92, 95, 103, 106, 108, 109, 119, 121, 124, 132

O

Objective function, 8, 9, 12, 72
Objective space, 9, 70, 72, 128
Optimization, 7
  Combinatorial optimization, 9, 10
  Continuous optimization, 9
  Decision variables, 86
  Discrete optimization, 9
Optimization algorithm, 10, 13
Optimization problems, 8, 9, 31

P

Pareto front, 69, 115, 129
Pareto optimal solution, 10
Population-based methods, 15, 34, 133

R

Research, 12, 13

S

Search space, 12, 70, 92
Search-Based Software Engineering, 3, 4, 23, 31, 32, 131, 134, 149, 165, 166
Shake, 72
SMCP, 4, 7, 13, 15, 17, 18, 20, 26, 32–34, 38, 75, 90, 131–134, 149, 152, 153, 158, 166, 167, 169, 170, 172, 174, 177, 179
Software Development Life-Cycle, 4, 5, 31, 32, 134, 150, 151, 166, 179
Software Engineering, 3, 26, 31, 134, 149, 166
Software maintenance, 5
Split operation, 81, 95
State of the art, 13, 15, 16
Swap operation, 78, 87, 95
Symposium on Search-Based Software Engineering, 31

T

Trajectory-based methods, 15, 86, 131–133

V

Variable Neighborhood Search, 34, 45, 46, 49–52, 54–56, 59, 64, 66, 75, 131
  Basic Variable Neighborhood Search, 50, 53, 66
  General Variable Neighborhood Search, 45, 50, 54, 55, 66, 69, 72, 102, 107–113, 119, 121, 124, 126–128, 131
  Multi-Objective General Variable Neighborhood Search, 45, 56, 57, 68, 69, 72, 102, 114, 116, 124, 127, 128, 131
  Multi-Objective Shake, 56, 57, 72, 116
  Multi-Objective Variable Neighborhood Descent, 34, 37, 56–58, 69, 75, 117, 128, 171
  Multi-Objective Variable Neighborhood Search, 45, 46, 54–56, 69, 95
  Shake, 50, 53–55, 57, 73, 110, 112
  Variable Neighborhood Descent, 34, 36, 45, 50, 53–55, 58, 59, 64, 66, 67, 102, 103, 105, 124–126, 131, 170
Vector of objectives, 9, 10
Very Large Scale Neighborhood search, 38

W

Wilcoxon's Signed Rank test, 125, 127