UNIVERSIDAD VERACRUZANA

Artificial Intelligence Research Institute

HEGA a genetic algorithm for solving the Software Module Clustering Problem

Thesis

In fulfilment of the requirements

for the degree of

Master in Artificial Intelligence

Submitted by

Juan Carlos Benjamín Somohano Murrieta
Software Engineer

Supervised by

Dr. Efrén Mezura Montes
Dr. Marcela Quiroz Castellanos

Xalapa, Ver. November , 2023

«In my experience, there is no such thing as luck»

- Obi-Wan Kenobi

Dedications

To my parents Juan Carlos & Llely,

To my brother Rafael,

To my three brothers who watch me over from Heaven,

To my best friend Miguel, and

To all my friends and acquaintances.

Acknowledgements

I want to thank my family, for their support during this process that I started two years ago. Their patience and their love were, are and always will be my strongest foundations, this work is completely dedicated to you.

My special thanks and gratitude for Dr. Efrén Mezura Montes, I couldn't have asked for a better thesis advisor.

I also want to thank my co-advisor Dr. Marcela Quiroz Castellanos for her support, patience and reviews. Without her observations this research could not have been possible.

I want to express my gratitude to Dr. María de Lourdes Hernández Rodríguez, for her guidance, reviews, and suggestions that contributed to this document, as well as her help during my research stay at the Laboratorio Nacional de Informática Avanzada (LANIA).

My gratitute goes also to my friends, who have always ecouraged me in everything I've wanted to do.

I want to thank the Mexican Council of Humanities, Science, and Technology (CONAHCyT) for the scholarship given to me during my studies (CVU 1143737).

Abstract

Software Module Clustering is a time-consuming task that aims to create a visual description that matches the architecture defined for a system. Search Based Softawre Engineering (SBSE) refers to Software Clustering as an NP-Hard Problem. Which means that the search-space increases considerably as the size of the systems to cluster grows. This means that search techniques, such as evolutionary algorithms, can be used to find a solution. In this thesis a new genetic algorithm is proposed to solve the Software Module Clustering Problem.

A hybrid encoding scheme is selected as the core of the proposal. Two different crossover operators are used to exchange the information of the selected parents into the offspring. An ad-hoc mutation operator is proposed based on corrective clustering and the orphan adoption algorithm taken from the literature. Likewise the other proposals in the literature, Modular Dependency Graphs (MDG) are selected as the input for the algorithm designed.

The experimentation is conducted to evaluate the behavior of the proposed algorithm and its robustness. The MDG of eight different systems are used as datasets for testing the proposal. The results obtained show that the proposed algorithm, called Hybrid Encoded Genetic Algorithm (HEGA), presents an improval in the results and robustness compared to other algorithms proposed in the literature.

Resumen

El agrupamiento de módulos de software, o clustering de software, es una tarea laboriosa. Dicha tarea busca generar una descripción visual de un sistema que coincida con su arquitectura. La ingeniería de software basada en búsquedas (SBSE) cataloga este problema como NP-Duro. Es decir, que el espacio de búsqueda llega a ser muy grande conforme crecen los sistemas a agrupar, que se vuelve necesario utilizar técnicas de búsqueda como los algoritmos evolutivos para recorrerlo y encontrar una buena solución al problema en un tiempo relativamente corto. En esta tesis se propone un algoritmo genético para resolver el problema del clustering de software.

El algoritmo, llamado Algoritmo Genético de Codificación Híbrida (HEGA), está formado por una representación de soluciones que consiste en una cadena binaria y un vector de números enteros. Por lo mismo, se utilizan dos tipos distintos de operadores de cruza para cada una de las partes de la solución. Se propone, además, un mecanismo de mutación que está basado en el manejo de módulos omnipresentes tomado de la literatura. A semejanza de otras propuestas, HEGA tiene como entradas Gráfos de Dependencia Modular (MDG).

La experimentación llevada a cabo tiene el propósito de evaluar la robustez del algoritmo y comparar si, en efecto, hay mejoría en los resultados en comparación con las herramientas tomadas de la literatura especializada. Los resultados obtenidos muestran que HEGA, en efecto, obtiene mejores resultados y es más robusto que otras propuestas de la literatura que también utilizan algoritmos genéticos.

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence.
CO	Combinatorial Optimization.
CP	Crossover Probability.
DCT	Draco Clustering Tool.
DE	Differential Evolution.
EA	Evolutionary Algorithms.
ECA	Equal Sized Cluster Approach.
EDA	Estimation of Distribution Algorithms.
EP	Evolutionary Programming.
ES	Evolutionary Strategies.
FCA	Fast Clustering Algorithm.
GA	Genetic Algorithm.
HCA	Hill Climbing Approach.
HEGA	Hybrid Encoded Genetic Algorithm.

MaSMCP Many-Objective Software Module Clustering Problem.

MCA Maximizing Cluster Approach.

MDG Module Dependency Graph.

MG Maximum Number of Generations.

MoSMCP Mono-Objective Software Module Clustering Problem.

MP Mutation Probability.

MQ Modularization Quality.

MuSMCP Multi-Objective Software Module Clustering Problem.

NAHC Next Ascent Hill-Climbing.

OAM Orphan Adoption Mechanism.

PS Population Size.

PSO Particle Swarm Optimization.

SA Simulated Annealing.

SAHC Steepest Ascent Hill Climbing.

SBO Search-Based Optimization.

SBSE Search-Based Software Engineering.

SDD Software Design Description.

SE Software Engineering.

SM Software Modularization.

SMCP Software Module Clustering Problem.

1 Introduction

Nowadays, software systems require to have a systematic process to be developed correctly and ensure the quality in the final result. Software Engineering (SE) is a branch of computer science that focuses on the application of engineering principles during the life cycle of the development of a software system [1]. In the software life cycle, five main stages can be identified: Requirements, Design, Implementation, Testing, and Maintenance. Within these stages there is a considerable number of situations which require the software engineer to think in the most suitable way of balancing the different goals and expectations given to a specific project. During the software life cycle, the developer can think in questions such as *What is the minimum number of test cases to cover all features and branches in this system?* or *How should the system's components be divided to get the optimum traceability?*. These questions can be answered in the literature through empirical studies and specialized experimentation. However, there is a crucial aspect to consider which is the time that these experiments take to be completed through SE estimations alone. In recent years, there has been an interest in reformulating software development activities and tasks as optimization problems in order to help the developers find suitable solutions to these problems in a considerably shorter period of time [2, 3]. Search-Based Software Engineering (SBSE) is an approach that aims to apply Search-Based Optimization (SBO) techniques to solve the problems that can be found during the software development life cycle.

SBSE has had a particular interest in software design and testing. During the design stage, an important aspect to consider is the modularity of the system. This refers to the importance of partitioning the system's architecture into small groups, called modules, that implement similar functions and can be managed autonomously interfering the least possible with other parts of the system itself. A Module Dependency Graph (MDG) is a low-level visual representation of the relations between each component of a system. It usually represents the distribution of the files that form the source code of the system. The complexity of modularity relies on the number of artefacts that a system has. The higher the number of components there is, the bigger the search space to look for a suitable partition becomes [1, 2, 4]. Optimization problems can be divided into two main categories: (1) those in which the final solution is represented in real values, and (2) the ones that gather solutions represented in a discrete range of values. Examples of the latter are the Combinatorial Optimization (CO) problems. Within CO, it can be found a particular type of problem, the grouping problems. In these, a set of items needs to be partitioned into a collection of subsets or groups so that each item belongs to only one group. To seek the best possible distribution of such items represents a challenge difficult to solve. This difficulty increases as the number of items to group does [5].

Evolutionary Algorithms (EA) are stochastic search strategies that are inspired by natural evolution. These methods implement some of the concepts found in organic evolution, such as crossover, mutation and natural selection. In general terms, for an EA to be considered as such, six factors must be present: (1) A representation of the solutions, (2) A population of N potential solutions, (3) a fitness function, (4) a selection mechanism, (5) variation operators, and (6) a survival mechanism [5, 6, 7]. A Genetic Algorithm (GA) is one of the most popular EA used for solving optimization problems. This implementation represents each solution as a string of elements, such as bits, natural numbers, real numbers or even permutations. These representations are called encoding schemes, and a solution correctly encoded shall be referred to as a chromosome. The GA will base all of its computing on these chromosomes and the evaluation of the fitness of each member of the population. GAs are one of the most popular search techniques used in SBSE. The Software Module Clustering Problem (SMCP) has been an optimization challenge for which multiple EAs have been proposed. The literature shows that by using this type of algorithm, a good partition of a system can be found in a relatively short time. Through the years there has been an evolution on the approach given to the SMCP. There are authors that deal with the problem as a Mono-Objective one [8, 9, 10, 11], and others who represent it as a Many or Multi-Objective problem [12, 13, 14, 15, 16, 17].

In recent years, software engineering problems require optimizing multiple objectives that are often in conflict with each other. Within the MaSCMP category, two main approaches can be found: the Maximizing Cluster Approach (MCA), and the Equal Sized Cluster Approach (ECA). Both aim to solve the problem by maximizing the Modularization Quality (MQ) ratio. However, they have certain differences in the rest of the criteria to be considered, which will be discussed later in this work. The MQ ratio is a real value which represents the level of cohesion and coupling in the partition determined for an MDG [18]. The MCA is the most studied approach in literature, which leaves a niche to be exploited. In this thesis, a genetic algorithm for the automatic modularization of software systems is proposed. This method is capable of finding partitions for a system that have a good modularization quality considering an ECA approach.

1.1 Problem definition

Mapping the design decisions, such as modularity, into the source code of a software system can become a challenging task. Software Modularization (SM) aims to find the best way to group the components of the source code that have similar functionalities to, then, validate that the system is being implemented according to the designed modularity.To perform this task is time-consuming. Furthermore, SM increases its complexity as the number of elements and relations in a system grows, not to mention the problems that come out from adding new elements to a system that is already being implemented. SBSE deals with this problem with search techniques such as genetic algorithms. Nonetheless, the results are not absolute, and they should not be treated as such since the search space is enormous and a better solution might always be found. It is necessary to generate a solution that maximizes the modularization quality, which depends directly on the cohesion and coupling within the system. A correctly partitioned system can be more understandable and its maintenance becomes more straightforward [1]. It is true that not all systems are equally complex, which leads to scalability problems within the different algorithms used for clustering software. This means that a method that produces an optimal solution for a particular case might not have the same outcome with larger and more complex systems [19].

1.2 Research Proposal

This work proposes an Evolutionary Algorithm with a single-objective multi-factor fitness function for the segmentation of Module Dependency Graphs to find partitions with a high Modularization Quality value (obtained by the *TurboMQ* function proposed by Mancoridis) similar or higher than the results obtained from software clustering tools found in the literature. This proposal will be shaped by:

- A genetic algorithm that can explore and exploit the search space.
- A novel encoding scheme that assists the convergence of the algorithm by reducing the redundancy of the solutions.
- An objective function that considers the quality of the modularization, the level of cohesion and coupling in the MDG, the number of clusters and their size.

1.3 Justification

During the construction of any software system it is important to determine whether or not the source code reflects the decisions made in the design stage of the project (see Section 2.1 for further comprehension). The latter contributes to the quality of the final product in terms of *Mainaintability*[1].

A good software design, among other features, should divide the components of the system and group them properly. This has a direct impact on the implementation of the system. The SBSE is constantly in the quest of finding new methods and

[1]For further information related to the quality factors of a software system, see [20, pp. 398-414] and standard ISO/IEC 25010:2011 [21]

techniques to improve the execution of software engineering processes. For this purpose, many optimization algorithms are used, such as genetic algorithms, combinatorial optimization heuristic techniques, and grouping algorithms, among others. Although Evolutionary Algorithms (EA) have shown promising results, most of the works found in the literature present variations or the adaptiation of a similar encoding scheme to specific constraints and domains. However, modern software systems present a higher level of complexity, which leads to adapting the evolutionary approach with other local search techniques in order to find better solutions that the EA alone cannot provide. As systems grow in complexity, it becomes necessary to create better descriptions of their functions and relations so that the development team can create and maintain a product that meets the specifications, desires and expectations of the client and the final users.

SBSE constantly searches for new ways to solve SE problems and for that reason, many optimization techniques are used such as genetic algorithms, combinatorial optimization heuristic techniques, grouping algorithms, among others [2]. From the Artificial Intelligence (AI) viewpoint, combinatorial optimization has attracted considerable attention towards the solving of problems in a great variety of fields, from everyday-life situations to complex engineering problems.

Lastly, it is important to mention that the literature shows that most of the algorithms used for clustering software systems consider the difficulty of restricting a single fitness function with multiple constraints instead of dealing with the problem as a multi-objective optimization one. In the majority of documented cases in SE this increases the complexity of the decision-making process at the end of the computation to determine which of the found solutions is better than the rest [16].

1.4 Hypothesis

A Genetic Algorithm that implements a Multi-Factor fitness function, and a hybrid encoding scheme, with an Equal-size Cluster approach applied to software module clustering, can find partitions for Module Dependency Graphs with modularization quality and cluster number values higher than other tools that implement the same type of algorithm, as well as smaller values in the difference between the size of the biggest and the smallest cluster within the same partition.

1.5 Objectives

1.5.1 General Objective

To design a Genetic Algorithm capable of exploring a discrete search space to find partitions for Module Dependency Graphs with a Modular Quality equal to or higher than the results shown in proposals documented in the specialized literature.

1.5.2 Specific Objectives

1. Gather information on the *state-of-the-art* on search techniques, with emphasis on genetic algorithms proposed for solving the SMCP.

2. Collect testing data sets used in the literature as well as software module clustering tools that implement search algorithms.

3. Determine the most suitable variation operators and parameter values in order to adjust them to the proposed objective function for generating results that meet the desired conditions.

4. Compare the results obtained from the genetic algorithm with those computed with proposals extracted from the state-of-the-art.

1.6 Contributions and knowledge dissemination

The main contribution of this research is a new proposal of a genetic algorithm with a hybrid encoding scheme and a mutation operator that considers the clustering of omnipresent modules. However, two colateral works derived from the research made on encoding schemes and their performance applied to combinatorial optimization problems are the following:

- Juan Carlos B. Somohano-Murrieta, Héctor Gabriel Acosta-Meza, Efrén Mezura-Montes; **GradCAM integration in the evolutionary design of Dense Convolutional Networks for medical images classification**; International Seminar of Computational Intelligence (ISCI 2022); Tijuana, Mexico (online).

- Juan Carlos B. Somohano-Murrieta, Efrén Mezura-Montes, Marcela Quiroz-Castellanos; **A new solution encoding scheme for solving the Flexible Job-Shop Scheduling Problem**; Congress of Evolutionary Computation (CEC 2023); Chicago, USA.

1.7 Document structure

This document is organized as follows:

- Chapter 1:The first chapter sets an initial background, and statements of the problem; and describes the solution to be evaluated.

- Chapter 2: This chapter describes in detail the background of software clustering, and how artificial intelligence is used to solve this problem

- Chapter 3: The third chapter describes the structure of the proposed algoritm, its encoding, the objective function and the operators and mechanisms used for the evolutionary process.

- Chapter 4: This chapter presents the results obtained from the experimentation, as well as their analysis comparing them with the proposals taken from the literature.

- Chapter 5: The last chapter states the conclusions of this thesis and exhibits the areas in which future research can enrich this work.

2 Theoretical Framework and Background

In this chapter, a theoretical framework is presented to familiarize the reader with the key concepts on which this thesis is based. This chapter introduces the main fields of this research shall be introduced: Software Design with an emphasis on Modularization, Evolutionary Algorithms, and Search-Based Software Engineering. Furthermore, this chapter also presents a review of the *state-of-the-art* on genetic algorithms applied to software module clustering.

2.1 Software Design

According to standard *ISO/IEC/IEEE 12207:2017(E)*[22], the *Software life-cycle* can be described as a set of processes that describe the development of a software system from the moment it is conceived as a necessity for solving a particular problem until its disposal. Software engineering is the application of systematic methods, processes, scientific and technological knowledge and expertise to the life-cycle of a software system [1, 23]. According to [24], software engineering has four main disciplines in charge of the four stages within the *software life-cycle process.* These disciplines are shown in figure 2.1.

The design of a software system is one of four commonly accepted stages that form the *software life-cycle process.* The software design can be seen as a set of two aspects:

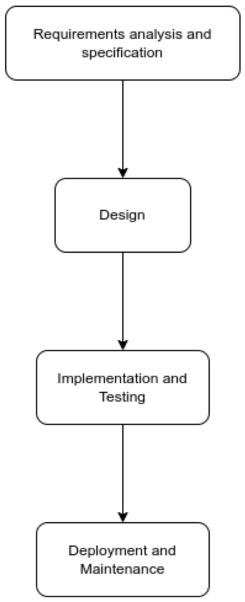

Requirements analysis and specification
Design
Implementation and Testing
Deployment and Maintenance

Figure 2.1: Software engineering disciplines applied to the software life-cycle described in [24]

(1) the process, and (2) the product. The process involves analyzing the requirements elicited in a previous stage in order to set out the internal structure of the system, its attributes and how they will be related at different levels of detail. These levels of detail describe source code (functions, modules, and classes); files and subsystems; and packages and components [25]. The main goal of the software design is to help the developers to create a detailed description of the system. This description shall be used as a guide for constructing the system and a reference for future engineers that helps to understand the functioning and make the maintenance process more straightforward. This description is called a Software Design Description (SDD) [26]. One important aspect that gives importance to the SDD is *communication*. Communication is a process that involves at least five main aspects: (1) Sender, (2) Receiver, (3) Message, (4) Channel and (5) Code. The sender is the source of the message, and the receiver is the destination of the latter. The code helps the sender and receiver to communicate, the sender encodes the message and the receiver decodes it with the same reference so that both understand the same thing. However, without a proper channel, the message cannot reach its destination. The SDD works as a channel for the software design team to communicate with the rest of the development team. Furthermore, an

understandable SDD will help to communicate in the future with the people in charge of the maintenance process.

The design stage is often underrated since a design specification is intended for software engineers to understand and, thus, is not an essential part of the final product for the client. Nonetheless, if a system is not correctly designed the consequences, which will be interpreted as errors, will be noticed anyway, whether there is an SDD or not. The difference shall be that with a proper SDD, the engineers will be able to identify the problem more easily than without one [27].

Software design is, essentially, a problem-solving task, Budgen [27] expresses that the design process can be reduced to describing the way in which a certain requirement shall be met. The design specification is deeply related to the abstraction and how the designer interprets the problem and its solution. So the design can only be judged as useful or not by the context of the problem itself. However, according to McGlaughlin [28] there are some aspects to consider to generate a good software design. For him, a good design should:

- Implement all explicit requirements contained in the previous analysis and must adjust all implicit requirements desired by the client.
- Be a guide readable and understandable by all the people that build, test, and give maintenance to the system.
- Give a complete idea of the internal functioning of the system, the domain it is located in, and its behavior.

2.1.1 Software Modularity

Divide et impera or *divide and rule*; this phrase is as ancient as mankind. It has been proven over hundreds of years that dividing a larger force into small and isolated groups exponentially increases the probability of victory. This strategy can be applied to almost any situation. Complex problems are divided into smaller concerns in order to be cleared out by a set of solutions, one for each concern. In the field of software design, it is important that a design specification is understandable. Making a detailed description of the structure of a system becomes glaringly challenging as the complexity of the latter increases [20]. One way to deal with complexity is to divide a system into smaller groups called *modules*. This principle is called *modularity*, and it is an effective mechanism in software design. A module is an implementation unit that arranges a set of elements with common responsibilities. This means that each module unit should include components, classes, and methods that assemble to meet the specification of a requirement or related requirements [1, 29].

Coupling and Cohesion

Although any system can be divided into smaller components, the designer has to be careful of grouping the elements that have similar objectives. The modularity of a system is directly affected by the similarity that exists between the components inside a single module, and the difference between a single module and the rest. The direct relationship between the components of a module is called *cohesion* and the distance among different modules is the *coupling* of a system. The higher the cohesion of a module the better. On the opposite side, as the coupling between modules is reduced, the modularity of a system will increase [20, 30, 31].

According to Budgen [27] and Pressman [20], it is highly desirable that a system's modularity has an increased level of functional cohesion. This means that all the elements inside a single module contribute to the execution of the same task. On the other hand, logical cohesion is hardly recommended. This type of cohesion refers to elements that perform logically similar operations but internally involve a different process. In terms of coupling, there are also different types and each one has its own desirable rate. In general terms, different elements that require sharing data and functions to complete a task should be placed in the same module, this is called *data coupling*. Nonetheless, the parameters of one module are not supposed to be known or shared by other modules and should not be required to determine their actions, this can be called a *Common-environment coupling*. The literature describes other types and levels of coupling in a system, according to the different levels of detail that an SDD has[1].

[1]For a more detailed description of the different levels of coupling nd cohesion in a system refer to [27, pp. 75–80], [20, pp. 225–229], and [30, pp. 50–67]

2.1.2 Software Clustering

Now that the concept of modularity and its importance in the understanding and management of the system have been described, the process that involves the execution of this property of software design will be explained. Clustering refers to the activity of grouping software artifacts. At different levels, the clustering might be performed over functions in the source code and group them into classes, files into packages and components, or packages and classes into subsystems. This thesis shall focus on the clustering of files into components. The reason for this is that the clustering process is performed on the source code of the system, either because the system is going through maintenance and there is no design document, or because the developers need to evaluate if the code really meets the design and analysis performed on the system [25, 32, 33].

The Software Module Clustering Problem (SMCP), sometimes called Software Modularization Problem, has become an active research area for software engineers. There are several algorithms proposed in the literature for solving this problem, some of which shall be discussed later in this document. Nonetheless, it is important to describe the basic process, since all the proposed algorithms aim to automate this activity [34]. According to Sarhan et. al [35], the software clustering process can be divided into five stages: First, a factbase needs to be extracted from the product. This factbase is the input for any clustering process. In this case, the relationships between the different artifacts of a system. This information is extracted from the source code. The similarity evaluates the closeness between the different elements in the factbase. Once the similarity is evaluated, the clusters need to be created by a suitable algorithm, there are multiple algorithms that have been proposed, some of which shall be described further in this document. Finally, the results obtained are compared and measured according to the modularity of the outcome.

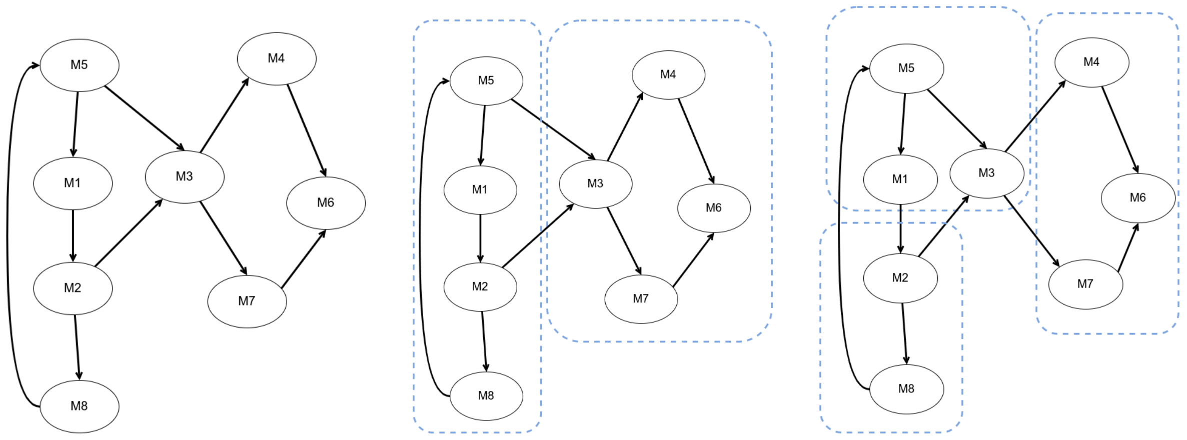
The clustering process of a system concerns the quality of the software design. As mentioned previously, a modular design of a system is useful for simplifying its construction and maintenance. However, the evaluation of the quality of the design cannot be completed without the final product. The modularity of a system is evaluated in terms of its coupling and cohesion, and these two aspects can be measured in the clustering process so that an algorithm can differentiate between a suitable modularization and a non-suitable one. But for this to happen, it is necessary to have a way of representing and obtaining useful information about the relationships between the elements of a system. These relationships and elements are obtained usually from the source code of the software [2, 25, 34, 36].

Module Dependency Graphs

An important part of the clustering process is the representation of the artifacts that need to be grouped. This representation will be useful for having a picture of the system and will make the interpretation of its structure easier. That is why it needs to be generated in a *language-free* way [10, 25]. During the design stage in the software development process, it is important to understand correctly how a system is divided in order to get the correct traceability between the architectural design and the distribution of the system in lower levels. One of the most common ways to understand the distribution of a system is by using directed graphs called Module Dependency Graph (MDG). In such graphs, each module of a system is represented by a node, and the relationships between those nodes are usually represented as directed edges [8].

Formally an MDG is defined as follows [37]: a graph $G = (M, R)$ is an MDG if is formed by two components M and R where: M is a set of named modules in a software system, and $R \subseteq M \times M$ is a set of ordered pairs $\langle u, v \rangle$ representing the relationship between the module u and the module v. The MDGs can be classified as weighted and unweighted. The first type has an importance value (a weight) associated with each relationship between two artifacts, while the latter disregard this value. The MDGs are the input to every modularization algorithm. The graph representation of a system is helpful to understand its structure. The clustering process consists of dividing an MDG with $|V|$ modules into m non-overlapping partitions. As mentioned previously, the SMCP can be applied at a low level in the software process, concerning artifacts such as files and the relationships between them in terms of function calls and shared data [9, 10, 37].

The complexity of the clustering process is in distributing all the modules in a



(a) (b) (c)

Figure 2.2: An example of an MDG and two different potential partitions

suitable number of partitions, Figure 2.2 exemplifies this statement. A single MDG such as the one in Figure 2.2(a) can be divided into different partitions, such as the examples shown in Figures 2.2(b) and 2.2(c). Each partition has a different level of modularization, depending on the intra-cluster and inter-cluster relationships between the modules, and whether these relationships have an associated weight or not.

Omnipresent modules

An important aspect to consider during the clustering process is the presence of modules and classes that are related to several parts of a system. These *Omnipresent Modules* have a significant value since they are crucial parts of a system but cannot be easily classified into a single group [38]. An omnipresent module can be a provider or a client of the system. The latter means that it can either include only methods and processes called by other modules (such as code libraries) or depend entirely on the other parts of the system (such as a *main* module) [39, 40, 41].

The two most popular positions regarding omnipresent modules are either to remove them before the clustering process or temporarily ignore them and take them up again once the process is finished so that the process itself is not obstructed by them. One of the first techniques for dealing with omnipresent modules was proposed by Müller et al [38], they implemented an algorithm to classify these particular modules as noise in the recovery of software architecture. This algorithm removes the noise from the architecture so that the system is recovered as close as possible to the desired structure. Even hough the removal of omnipresent modules increases the quality of the obtained clustering [42], some authors such as Zihua and Tzerpos [39], Patel et al [43], Kobayashi et al [44] claim that omnipresent modules are important in the structure and decomposition of a system's architecture. These authors mention that a clustering algorithm should be tolerant of omnipresent modules. Instead of removing the omnipresent modules from the modularization process, proposals have been made in recent years to give them their own grouping process, either by separating them into their own cluster or by reintegrating them at the end of the process into the cluster with which they are most related. However, one of the main problems when temporarily removing omnipresent modules and reinserting them at the end is to determine which cluster depends on each omnipresent module the most [45].

Tzerpos and Holt [46] created one of the first clustering algorithms to consider omnipresent modules, this algorithm is called Algorithm for Comprehension-Driven Clustering (*ACDC*). The authors proposed to identify the omnipresent modules such as libraries and group them in a separated cluster so that the modularization process is not affected by them. In 2005, Zhihua and Tzerpos proposed the FICABOO framework, which is a mechanism based specifically on the reinsertion of omnipresent modules when combined with a detection method [39]. The FICABOO framework is based on the *Orphan Adoption algorithm* proposed by Tzerpos and Holt [47]. This algorithm was initially proposed for the insertion of new modules (orphans) into the system or to reorganize the modules' distribution during an upgrade or in a maintenance stage. The process consists of taking a module and inserting it in the cluster that maximizes the quality of the modularization. The same principle can be applied to omnipresent modules. If these are extracted from the initial partition of an MDG, they can be reinserted following this algorithm. This is how the FICABOO framework works. A variation of FICABOO is implemented in the algorithm proposed in this document. The details will later be discussed in Section 3.3.4.

2.2 Evolutionary Algorithms

In the rubric of computational intelligence, evolutionary algorithms have been one of the most popular methods for solving different types of optimization problems. This type of algorithm has proven to be particularly reliable for NP-Hard problems, those which have a solution search space big enough to make an exhaustive search impractical. Evolutionary Algorithms (EA) are a computational allusion to biological evolution. This technique is based on the postulate of Charles Darwin on the evolution of species through natural selection. EAs take advantage of adaptative populations, which means that instead of working on a unique solution, the EAs generate and modify a set of potential solutions hrough a fixed number of iterations. Likewise the natural process of evolution, each iteration involves a process of selection, reproduction, evaluation and survival [56]. Between the 1960s and the 1970s three different search algorithm implementations were developed: Evolutionary Programming (EP), Genetic Algorithms (GAs), and Evolutionary Strategies (ES). These three algorithms were initially treated independently. It was not until the late 1980s and early 1990s when the term *Evolutionary Computing* was properly coined. Since the 1990s, EP, GA, and ES have exchanged ideas and techniques. David Fogel [50] expresses that these three algorithms have coexisted and interacted so closely that it is no longer possible to differentiate one algorithm from another in terms of only the representation, the variation methods, the selection mechanism, or any other factor. Nonetheless, their main differences now rely on their structural behaviour and overall performance when they are applied to a particular problem. For instance, ES models are strictly based on the extinction of the least fit individual, while the GA systems are commonly generational. Apart from the variants previously mentioned, there are other more recent algorithms, such as Differential Evolution (DE), Particle Swarm Optimization (PSO), and Estimation of Distribution Algorithms (EDA). Although PSO does not implement an evolutionary process; since rather than evolving, particles move across the space; in terms of algorithm design, it has enough elements to be considered within the EA family [57], but it is classified as a Swarm Intelligence approach.

In general terms, a typical evolutionary algorithm has four main characteristics:

(1) All of them are population-based, as mentioned previously in this section, the EAs iterate over a set of solutions rather than a single one; (2) They are stochastic, which means that randomness has a crucial role in all the mechanisms within the models; (3) They all have a survival mechanism, by evaluating an objective function each solution is determined to be more suitable than other, following the evolutionary schema only those solutions that are the most suitable will prevail; and (4) All of them can be used in a wide variety of problems without changing their basic flow, meaning that these are meta-heuristic techniques.

2.3 Genetic Algorithm

The Genetic Algorithm is a search technique proposed by Holland in 1975. It is one of the most popular EAs, it has been widely used for solving a large number of optimization problems. It emulates the principles of the Darwinian Theory regarding the survival of the fittest [56, 58]. Likewise other EAs, a GA is population-based and requires a selection method, crossover and mutation operators and a survival mechanism. The core of a GA relies on its generational flow; typically in a GA all members of the current population are dismissed and the offspring remains as the new population for the next generation. However, elitism can be also applied to a GA. In this case, the best solution of the current population is kept and reinserted into the new population. The latter guarantees convergence assuming infinite time [50, 56, 59].

2.3.1 Encodings

The encoding in a GA is the genetic representation of the potential solutions for a particular problem. An important aspect of an encoding scheme is the scope coverage. This means that the encoding of a GA should be able to describe the search space and describe all feasible solutions. Correctly representing the solutions is crucial in any GA. The encoding scheme has the responsibility of determining the characteristics that the algorithm will use to move across the search space [60].

Originally the GA only used binary representations, a string of bits that represented the main features of a solution [50]. Nonetheless, through the years other schemes that are equally acceptable have been included in the compendium of available representations. The selection of the encoding depends strictly on the problem for which the GA is to be designed. For example, binary representations are usually used when the problem requires to be interpreted in terms of *True/False* values, while real-valued encodings are mostly used when the solution is intended to be shown in a continuous space. According to Ronald [60], an encoding should:

1. Allow the exchange between genotype and phenotype in the minimum number of steps so that the computational cost does not increase considerably when estimating the fitness value.

2. Generate feasible solutions, if this is not possible in all cases, then the algorithm must implement penalty strategies or repair mechanisms.

3. Reduce the isomorphism within the solutions as much as possible. This means, for instance, dealing correctly with multiple genes that refer to the same phenotype (redundancy).

4. Represent the problem at a proper level. It should be possible to represent a complete set of potential solutions only with the rules and elements of the encoding using values taken from an alphabet correctly defined.

More recently, there have been multiple proposals to combine different encoding schemes. These hybrid representations are used often in the industry to solve problems whose complexity level does not allow a simple representation to reach an acceptable near-optimal solution. It is useful for partitioning a problem a represent different aspects of the solution. However, conventional variation operators cannot be applied in the same way. Variation operators (especially mutation) must be altered in order to diversify a gene depending on the coding of that gene [61].

2.3.2 Selection mechanism

As part of the evolutionary process, new solutions must be generated from the current population. It is necessary to determine which individuals will procreate the next generation. Since the purpose of evolution is to improve the currently available solutions, the fittest members have more relevance and the selection mechanisms have the inclination to prefer them over the rest. However, likewise in nature, all members should have the possibility to reproduce. That is why a stochastic component is usually added to this process. The latter refers to two important aspects of selection in GA: *Selection pressure* and *Population diversity*. A high selection pressure will tilt the search into the fittest values, this might eventually diminish the diversity of the population. However, a small pressure might cause the search to be fruitless. It is important that the selection mechanism finds a balance between pressure and diversity, and this is directly related to the encoding scheme used [14, 60, 61, 62]. In this section two of the most popular selection mechanisms are presented in general aspects. For a more detailed explanation of these and other selection mechanisms please refer to [63, pp. 117–121, & 163–192], [7, pp. 80–87], and [62].

Tournament selection

In this mechanism, a group of μ individuals is selected from the population. These selected solutions shall be evaluated according to the objective function. The fittest one will be selected as a candidate for the crossover stage. The process shall be repeated until the desired number of parents is selected. This mechanism has the advantage that neither the worst solutions will be selected nor the best individual shall dominate above all [64]. There are two versions of this selection mechanism: *Deterministic*, and *Stochastic*. In the first one, the individual with the best fitness is always selected, whilst in the latter, the fittest individual shall be selected with a probability p if and only if $random(U(0,1)) \leq p$, if this condition fails, then the next k-th best individual shall be selected with the same probability.

Proportional selection

In contrast with the previously-mentioned technique, in proportional selection, the probability of an individual being chosen relies on the contribution of its fitness value to the total fitness value of the population. This means that while the fittest individuals have more probabilities to be selected, the least fit ones are less likely to be chosen. However, the probability of the latter is never 0. This value leads to obtaining the expected number of copies of the individual to be selected for crossover. The probability value of a selection candidate can be obtained with equation 2.1

$$p_{sel} = \frac{f_i}{\bar{f}} \tag{2.1}$$

where f_i is the fitness value of the individual and $\bar{f}$ refers to the average fitness of the population.

There are multiple implementations of this model, such as the Monte Carlo, also referred to as the Roulette method [65], the Stochastic Remain [64, 66], and the Stochastic Universal Selection [63, p. 120; 57, p. 84; 62, p. 71].

2.3.3 Variation operators

The variation operators are another crucial part of any GA. Their objective is to ensure that potentially-good areas are found in the search space (exploration) and are used up as much as possible (exploitation). Eiben and Smith [57] express that the variation operators depend on the selected encoding scheme. The *Recombination* and *Mutation* mechanisms are used to find new solutions that lead the algorithm to a near-optimal state. The convergence of the algorithm is directly affected by these operators. Since genetic algorithms were initially conceived to be binary encoded par excellence, the variation operators that are most commonly used (and subsequently adapted) deal with the information of each chromosome at a low level.

Recombination

During the crossover of two selected solutions, called parent solutions, their information is combined to create a new individual; a new potential solution that ought to be different from both parents but keeping their phenotype characteristics encoded at a genotype level. This means that, during recombination, each new solution should not have additional information apart from the inherited by the parents. After randomly choosing the two parents previously mentioned, the crossover is applied with a probability $P(cr)$ called the *crossover/recombination rate*. There are different crossover methods, Umbrakar and Seth [59] made a review on crossover operators used for GAs. Among others, they mention some of the most common and simple mechanisms such as the *k-point* crossover, which takes k random positions of each parent solution and combines the information of both parents located within those positions; or the *uniform* crossover, which ensures that the information included in the offspring is transmitted by swapping the bits of both parents with a uniform probability $U(0, 1)$.

Mutation

Once new solutions are generated, the mutation is applied with a probability $P(m)$ called the mutation rate. This process typically changes the information in the genotype of a solution. The latter encourages the exploration of new areas within the search space. Mutation can be seen as a background operator whose primary objective is to ensure the variety of solutions [50, 58, 63]. There is an important concept for mutation operators which is *disruption*. This is the ability of the operator to alter the features of a solution. Rather than radically changing the fitness value of a chromosome, it should mainly vary its characteristics. That is why in GAs it is common to use a low $P(m)$, since this guarantees that the final offspring solution shall preserve the majority of its ancestors' information [50, 57, 63].

2.3.4 Other components of the GA

Replacement

Similar to the selection process, the replacement or *survivor selection* mechanism needs to identify the fittest individuals based on the objective function evaluation. This follows the principles mentioned at the beginning of this section regarding the survival of the fittest. A genetic algorithm is often generational, this means that every individual in the current population is discarded and the offspring remains for the next generation. However, there are some cases in which the best solution is kept and passed on to the next generation in order to ensure the convergence of the algorithm.

Initialization

The creation of the initial population in a genetic algorithm is performed typically by random sampling of a variable. This means that each gene of the chromosome is produced in a repeated stochastic process. It is uncommon to design GA to *assist* the initial population to converge towards good solutions [7, 57, 63]. However, real-world optimization problems get more complex every day, and sometimes it becomes necessary to optimize the performance of the GA. For instance, if an algorithm is supposed to optimize the total work distribution among several machines with multiple constraints (*Optimal Job Scheduling*), repairing solutions in the algorithm might represent a considerable computational cost. Therefore, the algorithm can be encouraged to generate only feasible solutions even from the initial population [67, 68].

Termination criteria

The standard criterion for terminating the execution of a genetic algorithm is the maximum number of generations. However, there are cases in which the algorithm can have other termination criteria. For example, the genotype diversity measure of the population, which allows to determine whether the average value of the fitness of the population is improved significantly with respect to previous results or not. Eiben and Smith [57, p. 34] mention that the real stopping criterion of a GA is reaching the optimum value. Nonetheless, this condition is highly unlikely to be reached, since real-world optimization problems typically do not have a known single optimal solution.

2.3.5 Standard Genetic Algorithm

Algorithm 1 includes the pseudocode of a standard GA.

2.4 Search-Based Software Engineering

Search-Based Software Engineering (SBSE) is a research field that has been gradually gaining popularity. This approach focuses on solving software engineering problems through Search-Based Optimization (SBO) algorithms [2]. During the development of

Algorithm 1 Template of a Genetic Algorithm

Require: Population size Ps, number of generations G_{MAX}, crossover rate CR, mutation rate MR, fitness function f

Output: Fittest solution

```
 1: Initialize population $P$ with $Ps$ random solutions
 2: $g \leftarrow 1$
 3: $best \leftarrow None$
 4: while $g < G_{MAX}$ do
 5:     for each $indv \in P$ do
 6:         if $f(indiv) \geq f(best)$ then
 7:             $best \leftarrow indiv$        ▷ Obtain the fittest element of current generation
 8:     Apply the selection mechanism to obtain a set of $Ps$ parents
 9:     while Offspring set $off$ is not full do
10:         Choose two parents $p_\alpha$ and $p_\beta$
11:         if $random(U(0,1)) \leq CR$ then
12:             Apply crossover operator to generate offspring $o_\varepsilon$ and $o_\gamma$.
13:         else
14:             $o_\varepsilon \leftarrow copy(p_\alpha)$
15:             $o_\gamma \leftarrow copy(p_\beta)$
```

```
16:         for each $o \in \{o_\varepsilon, o_\gamma\}$ do
17:             if $random(U(0,1)) \leq MR$ then
18:                 Apply mutation operator to offspring $o$
19:             $off \leftarrow \{o\} \bigcup off$        ▷ Add element to the offspring set
20:     Evaluate all members of $off$
21:     $off \leftarrow \{best\} \bigcup off$        ▷ Preserve the best solution to the
                                                      next generation
22:     Remove the least fit element of $off$        ▷ elitism
23:     $P \leftarrow off$
24:     $g \leftarrow g + 1$
25: Evaluate all elements in final $P$
26: Return the fittest solution
```

a software system, there are several situations that can be formulated as optimization problems [36]. For instance, in the testing stage, a developer might want to know the minimum number of test cases to execute that cover the totality of the system's functions or at least the largest possible amount. Harman et. al [2] provided an overview of the applications and challenges of SBSE up to that moment. This is important because most of the literature found that was published in more recent years departures from this research. Harman mentioned that there are three main challenges when applying search algorithms to software engineering: (1) Solution representation, (2) Fitness evaluation, and (3) Change operators. the encoding of a solution is crucial for designing the algorithm and will affect directly its performance. The complexity of an SBSE algorithm can be seen in the difficulty of decoding a solution rather than the algorithm itself. The function to optimize is particular to each problem. And the definition of a fitness value can be quite a challenge since many problems in SE require a subjective appraisal such as the opinion and feedback of the users. Finally, one of the most difficult problems to deal with is deciding how to explore the search space. Each algorithm might look for potential solutions in different ways. Each type of algorithm has its own parameters and mechanisms and will find the best possible solution according to such elements. This is why choosing a particular algorithm is complicated and decisive. There are several fields within software engineering in which search-based algorithms have been applied.

Software testing is the most popular stage in which SBSE has been implemented. Implementing different techniques and automated mechanisms to optimize the test case generation process in order to meet qualities such as coverage, completeness, traceability, etc [69, 70]. The test case and test data generation are crucial activities in software development. While constructing a system, the developers intend to cover all the requirements explicitly obtained from the analysis of the problem and the ones desired implicit by the client and user. This is why it is important to test every single aspect of the system in all combinations of scenarios available. However, in large systems, this becomes a problem since testing every possibility might consume a considerable amount of time and resources. That is why in SBSE aims to find a balance between the test cases and the desired quality of the system [71, 72, 73]. Another topic area in which software engineers have applied SBO algorithms is software management, in specific areas such as project planning, or cost/effort estimation [74, 75, 76]. This is an important area not only in software development but in almost every sphere. Every project has a finite amount of resources, a deadline, personnel to be placed, and so on. Software projects are not different, and these resources require to be distributed and correctly used in order to obtain the most benefit out of them [77, 78].

Besides these two vital aspects of the software life cycle, according to the literature, [2, 3, 36, 79], software design and re-design assisted by search algorithms have gradually gained special attention in the past two decades, despite the latter is also visualized as a maintenance problem. Three of the most popular problems in this field solved by search techniques are the software architecture design problem, the software quality assessment, and the SMCP. Depending on the moment and the state of the software system, it can be said that SMCP can be both a design and a re-design problem.

2.4.1 Software Modularization Quality

Jaccard coefficient

In 2003 Saeed et. al [80] made a review on software clustering techniques and combined algorithms used in pure software engineering. They mentioned that the main property that SE algorithms looked at in the modularization of a system was the correlation between the different clusters. The authors mentioned various metrics such as the *euclidean distance*, the *Sorensen-Dice coefficient*, the *Jaccard coefficient*, and the *Camberra metrics.* However, they said that the Jaccard metric was the one that provided the most suitable results. These coefficients can be obtained with equation 2.2; where, given two entities ω_1 and ω_2, a represents all features present in both entities, b refers to all the features present in ω_1 but not in ω_2, and c refers to the opposite of b. Finally, for the evaluation to be completed, a fourth value d is needed to obtain. This value represents the features absent in both entities ω_1, and ω_2 but present in other entities in the same system. If this value is small enough then it can be omitted. However, if the number of absent features is as large as a or b and c, then an equivalent metric might be used, this equivalent metric is the pearson correlation coefficient ρ. The authors exposed that ρ is equivalent to $J(\omega_1, \omega_2)$ when defined as shown in equation 2.3.

$$J(\omega_1, \omega_2) = a/(a+b+c) \tag{2.2}$$

$$\rho = \frac{ad - bc}{\sqrt{(a+b)(c+d)(a+c)(b+d)}} \tag{2.3}$$

Even though these metrics prove to be useful, the complexity of the evaluation of the correlation/distance between clusters grows exponentially as the complexity of the system does so. Nowadays software systems present an enormous number of modules and dependencies, which makes them unsuitable to use an algorithm with just such metrics.

BasicMQ

Compared to other fields of software engineering, search-based software clustering is a relatively new research subject, first formulated by Mancoridis et. al between 1998 [37] and 1999 [9]. The authors mentioned that, as seen in section 2.1.1, the modularization level of a system is given by the relationships between each component inside the same cluster and the association degree each cluster has with the rest. Nonetheless, the problem was to define an equation to assign a value to the quality of the relationships between the artifacts of a system. Mancoridis et al. [37] proposed a way to measure the cohesion and the coupling of an MDG. They interpreted the relationships of the modules inside a single cluster as the *Intra-connectivity* value given by equation 2.4, and the coupling between two clusters as the *Inter-connectivity* of their relationships shown by equation 2.5

$$A_i = \frac{\mu_i}{N_i^2} \tag{2.4}$$

$$E_{i,j} = \begin{cases} 0 & if\ i = j \\ \frac{\varepsilon_{i,j}}{2N_iN_j} & if\ i \neq j \end{cases} \tag{2.5}$$

where N_i are the number of modules in the cluster i, μ_i represents the number of the intra-edge dependencies (i.e., the number of directed edges between the modules in the graph), the maximum number of intra-edge dependencies is given by N_i^2, and $\varepsilon_{i,j}$ represents the number of dependencies between clusters i and j. With these two equations, Mancoridis defined the value of the modularization quality as the result of adding the values of each cluster's intra-connectivity and the total amount of inter-connectivity in the partition. Formally the value of the Modularization Quality (MQ): Given a Module Dependency Graph divided into k clusters, the Modularization Quality is evaluated through equation 2.6.

$$MQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j} & if\ k > 1 \\ A_1 & if\ k = 1 \end{cases} \tag{2.6}$$

This quality measurement shall adopt values between -1 and 1, where a partition with no intra-conectivity in its clusters shall have a quality value of -1 and one without inter-connectivity relationships will adopt an MQ value of 1. Neither of these two is desirable. This initial value of modularization quality will later be known as *BasicMQ*. This was the first approach proposed to evaluate the modularization degree as an objective function to be optimized with search algorithms. Mancoridis tested this proposal on different search algorithms and the results showed to be promising. This modularization function was also used by Doval et al. [8] as the fitness function of their implementation of a genetic algorithm for clustering software source code.

TurboMQ

In 2002 Mitchell [10] designed an improvement to *BasicMQ* that dealt with two main problems, the lack of support for weighted graphs that *BasicMQ* has, and the computational complexity of the evaluation process. This new proposal was named *TurboMQ*. Instead of dealing with the modularization value as a whole, this new proposal assigned an individual value for each cluster called *cluster factor*. According to Mitchell, the cluster factor CF_i of each cluster i is given by a normalized value obtained from the intra- and inter-cluster relationships. So, given an MDG partitioned into k clusters, the modularization quality $TurboMQ$ is obtained with equations 2.7 and 2.8. In this case, instead of dealing with the total amount of relationships, it evaluates each cluster individually, with μ_i for the relationships inside the cluster i and $\varepsilon_{i,j}$, $\varepsilon_{j,i}$ for the coupling value between the clusters i and j respectively.

$$TurboMQ = \sum_{i=1}^{k} CF_i \tag{2.7}$$

$$CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise \end{cases} \tag{2.8}$$

Mitchell proposed a third modularization quality function called *Incremental TurboMQ*, which relies on the similarity between two partitions. This means that every new solution obtained from a single partition will be evaluated in terms of the clusters that were changed and will keep the cluster factors of those that did not change. Mitchell proposed a complete search algorithm only to be used with this fitness function. However, this third objective function is rather uncommon to be found in the literature.

Structural similarity

Although *BasicMQ* and *TurboMQ* have been the most popular used in recent years, there are other metrics based on the similarity of the clusters in an MDG partition. One of the most recent is a metric proposed by Huang and Liu [81]. The authors designed a metric that considered certain aspects that, according to the authors, Mancoridis omitted when he designed the *BasicMQ*. These aspects are the presence of global modules and the unidirectionality of the directed edges between the clusters. The first one refers to those modules that are called from two different clusters and, hence, should be grouped in a separate cluster alongside the rest of the global modules. The second aspect mentioned by Huang and Liu is a direct property of the modularity of the software design, which stipulates that the data should be protected and shared in a single direction as much as possible for the software to be maintainable and meet the traceability attributes [1, 26, 63]. The authors formulated the similarity metric based on the structural similarity coefficient proposed by Huang et. al [81] as a quality function in community detection in network design. This metric was also used by Chenlong et. al [82] in their genetic algorithm for discovering communities in signed social networks. This metric considers a network as a weighted graph $G = (V, E, w)$ which is similar to a module dependency graph in software clustering. Huang and Liu adapted this similarity metric in order to consider the weights and responsibilities implied in a module dependency graph. The first step is to identify the global modules. Formally, Huang and Liu defined this metric as follows: *Given a modular dependency graph $MDG = (V, E)$ and a partition $C = (C_1, C_2, \ldots, C_m)$, a module $v \in C_k, k = 1, 2, \ldots, m$ is called a global module if it is called by modules in two different clusters appear from the one v belongs to, and v calls no other module.* According to the authors, this means that a global module v must satisfy the following conditions:

$$\exists u \in C_l, w \in C_j, l \neq j \neq k, \mid (\overrightarrow{u, v}) \in E, (\overrightarrow{w, v}) \in E$$

$$\neg \exists x \in V \mid (\overrightarrow{v, x}) \in E$$

Once the global modules are identified, these are allocated in a new cluster called C_{global} and their relationships are removed from the original partition. So, a new partition $C' = (C'_1, C'_2, \ldots, C'_m)$ is generated. It is then that the similarity value between two clusters u and v is obtained with equation 2.9 and 2.10. Where $\Gamma_{in}(u)$ refers to the set of all edges that are inside the same cluster and $\Gamma_{out}(u)$ represents all the edges that come out of the u to the rest of clusters.

$$s(u, v) = \frac{\sum_{x \in \Gamma_{in}(u) \cap \Gamma_{in}(v)} w(\overrightarrow{u,x}) \cdot w(\overrightarrow{v,x}) + \sum_{x \in \Gamma_{in}(u) \cap \Gamma_{in}(v)} w(\overrightarrow{x,u}) \cdot w(\overrightarrow{x,v})}{\sqrt{\sum_{x \in \Gamma_{in}(u) \cup \Gamma_{out}(u)} \left(w(\overrightarrow{u,x}) + w(\overrightarrow{x,u}) \right)^2} \cdot \sqrt{\sum_{x \in \Gamma_{in}(v) \cup \Gamma_{out}(v)} \left(w(\overrightarrow{v,x}) + w(\overrightarrow{x,v}) \right)^2}} \tag{2.9}$$

$$w(\overrightarrow{u,x}) \cdot w(\overrightarrow{v,x}) = \begin{cases} > 0 & if\ \overrightarrow{u,x}, \overrightarrow{v,x} \in E \\ 0 & otherwise \end{cases} \tag{2.10}$$

Finally, this similarity value is calculated for every cluster and for the whole partition. These similarity values are normalized and the *Tightness* value is calculated. To obtain the modularization quality value as the sum of every Tightness value.

2.4.2 Software clustering algorithms

Mancoriodis initially proposed three different algorithms to deal with the SMCP: (1) An exhaustive search algorithm, (2) A Hill Climbing algorithm and (3) A Simulated Annealing algorithm. He also formulated the initial perspective of a Genetic Algorithm. However it was later, Doval [8] and Mitchell [10] who propose the genetic approach for solving the SMCP as a complement of this series of search algorithms proposed as part of a software tool called *Bunch*, which shall be described later on section 2.4.5.

Exhaustive search

The first search technique applied to SMCP was an exhaustive search algorithm [10, 37]. As the name indicates, it evaluates all potential solutions within the whole search space. This can be effective for small systems. Nowadays this technique is not suitable for the complexity of the systems. Even Mitchel himself mentioned that this technique is only useful for small systems since it has to search over a vast amount of solutions. In the words of Mitchell himself, this algorithm has been used mostly for debugging and testing the accuracy of the solutions found by other search algorithms.

Hill climbing

Another of the first proposed algorithms to solve the SMCP is the Hill Climbing Approach (HCA). It is a local search algorithm that starts with an arbitrary solution, in this case, a random partition of an MDG, and then attempts to find a better solution in a neighborhood that contains a threshold of a minimum number of different partitions. Each neighbor changes significantly from the current solution and the algorithm should evaluate all the MQ values in order to find the best of that neighborhood. If the change produces a better solution, the new solution is as well changed incrementally until no further improvements can be found [9]. There are two main variants of the traditional HCA applied to SMCP, which are the Steepest Ascent Hill Climbing (SAHC) and the Next Ascent Hill-Climbing (NAHC). The first one progressively creates a new partition based on a neighborhood created with a maximal neighboring partition (MNP) of the current partition. This MNP is created by evaluating all partitions and selecting the one with the highest MQ. On the other hand, the NAHC attempts to find a better neighboring partition by randomly evaluating through the partitions formed from the current one until it finds one with a larger MQ [83, 84]. A traditional implementation of the HC algorithm applied to SMCP can be seen in algorithm 2.

Algorithm 2 Hill-Climbing algorithm applied to SMCP

```
Require: $MDG = (M, R)$, $P_s$
Output: The best solution found $\theta$
 1: $P \leftarrow \{P_1, P_2, \dots, P_{P_s} | P_\eta = rndm(MDG)\}$
 2: $\theta \leftarrow \varnothing$
 3: for each $p_i \in P$ do
 4:     let $\beta$ be the neighborhood of $p_i$
 5:     for each $C_i$ in $\beta$ do
 6:         repeat
 7:             $C_{i+1} \leftarrow rndm(N(C_i))$
 8:             if $MQ(C_{i+1}) > MQ(C_i)$ then
 9:                 $currentBest \leftarrow C_{i+1}$
10:         until No better neighbor partition is found
11:         if $MQ(currentBest) > MQ(\theta)$ then
12:             $\theta \leftarrow currentBest$
```

Simulated Annealing

A compliment to the hill climbing approach proposed by Mancoridis was the Simulated Annealing (SA). This algorithm emulates the physical process of metallurgy in which an object's temperature is decreased while molding it until it reaches a desired state. The SA algorithm starts from an initial solution and improves it iteratively during the annealing process. This process ios based on a neighborhood of solutions and the algorithm selects the best one and compares its MQ to the current best solution. There are some factors that affect the selection of a potential solution. First the temperature of the system. In this case the value of the temperature t indicates the probability of selecting the best solution once the neighborhood has been evaluated. The temperature of the system is initially set t_0 and decreases with a constant ratio α as the algorithm iterates until it reaches the froze temperature t_f [10, 37, 85]. Algorithm 3 shows a classic implementation of SA.

Algorithm 3 SA algorithm applied to SMCP

```
Require: $MDG = (M, R)$,cooling ratio $\alpha$, initial temperature $t_0$, freeze temperature
    $t_f$, constant $r$, fitness function $f$
Output: The best solution found $\theta$
 1: $\theta_i \leftarrow rndm(MDG)$
 2: $t \leftarrow t_0$
 3: while $t > t_f$ do
 4:     for $r_i \leftarrow 1, r$ do
 5:         $\theta_{i+q} \leftarrow variate(\theta_i)$
 6:         if $MQ(\theta_{i+1}) > MQ(\theta_i)$ then
 7:             $\theta_i \leftarrow \theta_{i+1}$
 8:         else
 9:             $\delta \leftarrow \Delta MQ(\theta_i, \theta_{i+1})$
10:             $p \leftarrow U(0, 1)$                          ▷ Uniformly Random probability
11:             if $p < e^{-\delta/t}$ then
12:                 $\theta_i \leftarrow \theta_{i+1}$
13:     $t \leftarrow \alpha * t$                                  ▷ Adjust system temperature
14: $\theta \leftarrow \theta_i$
```

Swarm intelligence algorithms

In SBSE there is a special type of search algorithm that has been gaining popularity for solving optimization problems. These are the Swarm intelligence optimization algorithms. These algorithms base their solving on emulating the social behavior of simple organisms where some kind of knowledge emerges [86]. This can be explained with one of the most popular swarm optimization algorithms, the *Ant Colony Optimization*, a single ant is usually not considered as *intelligent*, but a whole colony yes. Through experience and proper communication, a whole colony can simulate intelligent behavior. There have been multiple implementations of swarm optimization algorithms applied to the clustering of software. Some of them are the firefly approach proposed by Mamaghani and Hajizadeh [87], the grey wolf algorithm of Kumar et.al [88], the artificial bee colony approaches of Chhabra et.al, and Amarjeet [89, 90, 91], and the PSO implementation of Bishnoi and Singh, and Prajapati [92, 93, 94]. Each algorithm has its own functioning, however, they all share a similar behavior based on exploring the search space with several agents and communicating the best solution found by them, until a desired condition is reached.

2.4.3 Clustering with genetic algorithms

The Genetic Algorithm is, without doubt, one of the most popular approaches given to the SMCP. There is an ongoing number of algorithms proposed in the literature [4, 8, 10, 12, 15, 16, 17, 81, 95, 95, 96, 97, 98, 99, 100, 101, 102, 103]. In the following lines some of the encodings and the operators that have been used in the different GAs proposed shall be described. One particular aspect that all of them share in common is that most of them follow the model proposed by Mitchell and Mancoridis, it is until recent years that different operators or encodings have been proposed.

Encodings

In the literature there can be found that the most used encoding implements a list of discrete integer values generally called Simple-Encoding [4, 8, 10, 12, 97, 100, 104]. In this representation, each chromosome is a list of natural numbers in which each position represents a module of the system, and the value is the label of the cluster assigned to that position. For instance, a chromosome with 3 clusters and 6 modules would be represented as follows:

$$S = 2, 2, 4, 3, 4, 2$$

This would be translated as three clusters with labels *2*, *3*, and *4*, and each cluster would be represented as follows:

$$C_2 = \{1, 2, 6\},\ C_3 = \{4\},\ C_4 = \{3, 5\}$$

In 2005 Parsa and Bushehrian [11] proposed an encoding based on permutations, it keeps the representation as a list of integers, but in this case the semantics of the encoding is different. In this encoding, each chromosome C is a permutation of N integers, where the i^{th} gene of the chromosome rather than representing a partition assignment, it holds a value p where $1 <= p <= N$. In order to assign the modules to a partition the values of the chromosome are evaluated against their position in the chromosome. If the value $C(i)$ is higher or equal to i then a new cluster is created and assigned the i^{th} module. If the later condition is not satisfied then the module i is assigned to the same cluster as the module with the same identifier as $C(i)$. This encoding can be represented in the following example:

$$C = 1, 5, 6, 3, 2, 4$$

This would be decoded into three clusters. These shall group the following modules:

$$C_1 = \{1\},\ C_2 = \{2, 5\},\ C_3 = \{3, 4, 6\}$$

Another encoding scheme was proposed by Praditwong [101]. The author proposes a grouping representation, in which each chromosome is defined as a list of variable lengths in which each module is grouped according to the partition assigned to it. For example, a candidate solution shall be represented as a list with three groups:

$$L1 = \{\{1, 3, 5\}, \{2, 4\}, \{6\}\}$$

This representation shows that the first cluster contains modules 1,3, and 5; modules 2 and 4 are assigned to cluster 2 and the remaining sixth module is grouped within cluster 3. This representation describes the distribution of the partitions with no or little necessity for decoding and repairing.

Tarchetti et al. [16] proposed an encoding based on binary strings, in which each gene i is a binary number representing the number of clusters associated with the *ith* module. According to the authors, this reduces the memory space required to allocate each chromosome, since each element of the array occupies only $\left\lceil \log_2 \frac{|V|-1}{2} \right\rceil$ bits of the binary string, where V is the number of modules in the system.

There are other encoding schemes such as the vector-based encoding [15, 96, 98, 105], which represent a solution as a sequence of operations listed in a $N-dimensional$ vector. However, these encoding schemes can only be used when there is a previously approved modularization, since they are used for re-modularization algorithms. Nonetheless, in this thesis, only the search algorithms used for unsupervised clustering are considered.

Operators

Selection mechanisms According to literature, most authors have an inclination for *roulette*, *tournament*, or Linear-Ranking [4, 8, 10, 11, 97, 98, 101, 104] as selection mechanisms. However, there are some cases where, given the multi-objective nature of the proposed algorithms only a single value is not enough to select the parents, in which PESA's mechanism[106], or an NSGA-II-based [14] selection mechanisms are used [12, 16].

Crossover The crossover or recombination operators are directly related to the encoding used for the solutions. They have the responsibility of exploiting the search space based on the exchange of characteristics between the selected parents. Since each representation has its own range of operators. For example, when using Simple-Encoding or a discrete representation in general a one-point recombination operator is used [4, 8, 10, 11, 12, 16, 97, 104]. When it comes to a grouping representation, Praditwong [101] proposed a crossover operator specially designed for grouping representations which is similar to a 2-point crossover. Here, a subsection of each parent is copied to the offspring in the same order (Parent1 to Child1 and Parent2 to Child2), the rest is exchanged from each parent to the opposite offspring. Finally, the chromosomes are repaired by deleting repeated elements.

Mutation In the literature, there is a large number of operators used for mutating the offspring generated in the crossover process, since the mutation operators have the purpose of exploring new regions of the search space. Used operators include: simple mutation [11, 97], uniform mutation[4, 8, 10], swaping [99], min-cut and neighbourhood-based operators [98], or even binary flip [16]. As can be seen, in terms of mutation, in the literature authors do not have a particular preference for a single operator. However, a special case is the GGA proposed by Praditwong [101], since, the author mentions that for this particular case, a mutation operator was not considered. The author's work shows that, empirically, the crossover operation already manages to explore and exploit the search space.

2.4.4 Fitness in genetic algorithms applied to the SMCP

One of the main problems when designing a search algorithm in Software Engineering is determining the fitness value and the function to be optimized. Recently Gupta et. al [32] performed a review on the fitness functions that have been used in genetic algorithms applied to the SMCP. Evidently, the most common value to be taken as the fitness function is the modular quality, either the *BasicMQ* or the *TurboMQ* proposed by Mitchell and Mancoridis. Nonetheless, Candela et al. [107] mentioned that just focusing on coupling and cohesion when evaluating the modularization quality of a partition is not enough. Indeed these are the two determining factors that affect the fitness of a solution. However, over the years there have been proposals that enlighten the need for other factors, such as the number of clusters, the number of modules inside each cluster, and so on. That is why the following section will describe the main approaches given to the clustering problem.

Clustering approaches

For many years the objective function and the modularity evaluation of a system were taken as synonyms. However, through the years this has been changing. Morsali and Keyvanpour [18] exposed that in recent years there have been three main approaches to solve the SMCP, these are, the Mono-Objective Software Module Clustering Problem (MoSMCP), the Multi-Objective Software Module Clustering Problem (MuSMCP) and the Many-Objective Software Module Clustering Problem (MaSMCP) approaches. The difference between these approaches, likewise in other NP-hard problems, is the number of criteria that the algorithm deals with.

Regarding the MoSMCP there are two sub-approaches that can be considered: the single-factor based and the multiple-factor based approach [32]. The mono-objective single-factor fitness function takes only the modularization quality usually obtained with equations 2.6, 2.7, or 2.9. However, there have been different proposals applied to the single-objective multiple-factor approach, which usually are based on ordered sums of values associated with different objectives in the modularization evaluation. A good example of this approach is the proposal of Harman et. al [108], which evaluates three aspects: the cohesion of the system, the coupling *unfitness*, and the granularity of the clusters; all of these have equal weights associated in a single function.

$$Coh(m) = \begin{cases} 1 & N(m) = 1 \\ \frac{\mathcal{A}(m)}{\mathcal{N}(m) \cdot (\mathcal{N}(m) - 1)} & otherwise \end{cases} \tag{2.11}$$

$$Cohesion(S) = \frac{\sum_{m \in S} C(m)}{K} \tag{2.12}$$

According to Harman, the cohesion of a single cluster is given by equation 2.11, where $\mathcal{A}(m)$ represents the number of dependencies that come from other modules into the module m, and $\mathcal{N}(m)$ represents the number of relationships that a module m has with the rest of modules. The cohesion of the whole system is a sum of the individual levels of cohesion as shown in equation 2.12. Harman defines coupling as a negative value since the similarity between the clusters should be minimal, and is given by the number of dependencies in a single cluster divided by the total possible number for the network.

Another example of a fitness function that evaluates more aspects than just the modularization quality is the one proposed by Prajapati and Chhabra [93], the authors implemented a simplified version of the *BasicMQ* equation but taking into consideration the number of modules and the total number of clusters that the system was divided into. This can be seen in equation 2.13, where MD_{intra} and MD_{inter} represent the total amount of dependencies inside each cluster and the total relationships between clusters respectively; NC and NM are the number of clusters and the total amount of modules to divide respectively; finally $\{\alpha, \beta, \gamma\} \in [0, 1]$ are constant relative importance values.

$$fitness = \left(\frac{MD_{intra}}{MD_{intra} + MD_{inter}}\right)^{\alpha} * \left(\frac{1}{NC}\right)^{\beta} * \left(\frac{NC}{NM}\right)^{\gamma} \tag{2.13}$$

In the literature, the Multi-Objective approach has been recently implemented, since software systems often need to be optimized in several aspects that affect their modularity. Within this category there are two main approaches to be found: the Maximizing Cluster Approach (MCA) and the Equal Sized Cluster Approach (ECA) [104]. Both aim to solve the problem by maximizing the MQ value. However, they have certain differences in the rest of the criteria to be considered. Both are multi-objective approaches that have the main goal of evaluating the quality of the modularization to get the maximum cohesion with the minimal coupling possible [10, p. 77]. MCA has the following objectives:

1. Maximize MQ
2. Maximize cohesion
3. Minimize coupling
4. Maximize the number of clusters
5. Minimize the number of isolated clusters

Cohesion and coupling, as well as the MQ function, evaluate the quality of the solution, the first objective maintains the distance between the modules inside a single cluster is maximum, the second one ensures that the similarity between different clusters is minimum. The third objective measures the relationship between the first two objectives and ensures it remains optimal. The fourth objective avoids all the modules to gather together in a single cluster but has the risk of creating isolated clusters, which means that a single module is placed in a single cluster without dependencies with the rest of the modules. The fifth objective is responsible for minimizing the number of isolated clusters.

On the other hand, the ECA approach does not look for minimizing the number of isolated clusters, at least not directly. Instead, it looks for reducing the difference between the number of modules inside the biggest cluster with the number of modules inside the smallest one. According to Praditwong [103], the number of isolated clusters in an ECA approach to be minimum is the outcome of satisfying the other objectives of the approach:

1. Maximize MQ

2. Maximize cohesion

3. Minimize coupling

4. Maximize the number of clusters

5. Minimize the difference between the biggest cluster and the smallest cluster

There is another multi-objective approach proposed by Sun et. al [109], in which the authors, apart from evaluating the modularization quality through *TurboMQ*, also evaluate the number of reversed edges. This last value helps define the unidirectionally of the cluster, just as Huang et al mentioned previously in their research [81]. They mention that this value is obtained with equation 2.14. Where $L\left(V_i, V_j\right)$ represent the inter-edge number between cluster i and j, and $L(V_i, \overline{V_i})$ means the inter-edge number between the cluster i and the rest of clusters; m denotes the total number of clusters.

$$f_{dir} = \frac{1}{m+1} \sum_{i=1}^{m} \min\left(L\left(V_i, V_j\right), L(V_i, \overline{V_i})\right) \tag{2.14}$$

2.4.5 Software clustering tools

There have been some implementations of the search algorithms described above, these implementations seek to give a moderate interface in order for the users to modularize a system in fewer steps. Three of these tools will be described here, one of them is the first tool ever to be proposed that implemented a genetic algorithm for clustering software systems. This tool is called Bunch (1998). It is worth mentioning that Bunch is still one of the most widely used tool for clustering and for comparing the results obtained from other implementations. The second clustering tool to be compared with the proposal documented here is called Draco Clustering Tool (DCT). In the literature, at the moment when this research is made, this is the most recent tool that is publicly available for clustering software systems that implements a genetic algorithm. The third and last tool is one that implements a deterministic algorithm and aims to find better solutions than those that implement a stochastic search. It is called Fast Clustering Algorithm (FCA). This clustering tool, as far as this research gets, is a recently deployed system that does not implements a genetic algorithm, thus the results obtained from its execution are suitable to be compared with the ones obtained from the proposal of this document given its stochastic nature.

The three tools were obtained and tested with the datasets gathered for this research. The proposal presented in this thesis shall be compared with the results obtained from the tool described in the following lines. Although the evolution of the solutions cannot be tracked without altering the source code of the tools, the final results can be presented in terms of the values measured in this work. This validates the ground truth generated by the tools mentioned. An important observation is that the documentation found for two of these tools does not mention a mechanism to deal with omnipresent modules.

Bunch

Designed initially by Mancoridis et. al [9, 37] alongside the Drexel University Software Engineering Research Group [2], Bunch Clustering Tool was the first *GUI-guided* [3] software tool created for clustering. Initially, it implemented an exhaustive search algorithm, a hill-climbing approach, and a simulated annealing mechanism. Later, Doval et. al [8] added to the algorithms portfolio the implementation of a Genetic Algorithm. This tool was initially tested with 17 different systems with a variety of modules that went from 13 nodes in the MDG extracted to 153 modules. The results were promising at the time. The tool was improved over the years, new fitness functions like *TurboMQ* [10] were added, and other variations of the HC algorithm as well. The authors initially mentioned that the presence of omnipresent modules is one of the limitations to Bunch's automatic clustering capability. Later, they added a functionality to isolate these modules in order to make the clustering process easier but it reduces the modularization quality value that can be obtained from a partition.

[2]`https://drexel.edu/cci/research/research-areas/systems-and-software-engineering/`

[3]A Graphical User Interface is the connection between the machine and the final user through visual elements such as buttons, displays, text, and instructions

Draco Clustering Tool

The Draco Clustering Tool (DCT) [16] is a newly designed clustering tool based on a multi-objective clustering algorithm that implemented an equal-sized clusters approach (ECA) using the modularization quality evaluated by *TurboMQ*. They used the values for calibrating the parameters of the algorithm that were proposed by Candela et. al [107] years before in the genetic algorithm they proposed. Given a software MDG $MDG = (V, E)$ with $n = |V|$, the Population Size (PS), Maximum Number of Generations (MG), Crossover Probability (CP), and Mutation Probability (MP) are described as follows:

- $PS = \begin{cases} 2n & if\ n \leq 300 \\ n & if\ 300 < n \leq 3,000 \\ n/2 & if\ 3,000 < n \leq 10,000 \\ n/4 & if\ n > 10,000 \end{cases}$

- $MG = \begin{cases} 50n & if\ n \leq 300 \\ 20n & if\ 300 < n \leq 3,000 \\ 5n & if\ 3,000 < n \leq 10,000 \\ n & if\ n > 10,000 \end{cases}$

- $CP = \begin{cases} 0.8 & if\ n \leq 100 \\ 0.8 + 0.2(n-100)/899 & if\ 100 < n \leq 1,000 \\ 1 & if\ n > 1,000 \end{cases}$

- $MP = \frac{16}{100\sqrt{n}}$

DCT implements the Non-dominated Sorting Genetic Algorithm (NSGA-II) in order to perform the evolution of the population with a multi-objective approach. For further details refer to the original paper and the algorithm's definition in [14].

Fast Clustering Algorithm

The Fast Clustering Algorithm (FCA) proposed by Teymourian et. al [19] to cluster large-scale systems. This method aims to maximize cohesion and minimize coupling by using a matrix with neighboring scales, in which the values of the different related modules are shown when being neighbors inside the same cluster. After creating this matrix, the values are normalized, since, according to the authors, when the number of modules is large the total number of dependencies also grows. In order to attend to this, the effect matrix is created, which normalizes the neighboring matrix and after that, the nodes are ordered in a descendant way. The last n nodes are those with the highest probability of belonging to another cluster if their effect value is below a certain threshold.

To evaluate the modularization quality three formulas were used. The first one is a modification of *TurboMQ* mentioned also by Mamaghani et. al [87]. The authors mention that *TurboMQ* solves the two main problems related to *BasicMQ* which refer to the management of weighted MDGs and the computational complexity of the evaluation. This value is obtained with equation 2.15, here μ_i is the number of dependencies inside the same cluster and $\varepsilon_{i,j}$ is the number of dependencies between cluster i and j.

$$\begin{gathered} TurboMQ = \sum_{i=1}^{k} CF_i \\ CF_i = \begin{cases} 0 & if\ \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{j=1}^{k}(\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise \end{cases} \end{gathered} \tag{2.15}$$

The second and third metrics do not affect directly the algorithm. Nonetheless, they were used to evaluate the effectiveness of their proposal from the experts' point of view. The second metric used to evaluate the quality of the modularization was the *MoJoFM* value, which is used to compare the generated clusters with those pre-established by an expert. This metric can only be applied during a re-design activity in the maintenance stage of the software life-cycle. The values are constrained between 0 and 100, the higher the value, the closest the distance between the clustering generated by the computer and the expert opinion. The third and last metric is the *cluster-to-cluster* coverage (C2C), which evaluates the accuracy at a component level between an automatic-generated cluster and another considered as the objective to be reached (ground-truth) [4].

Their proposal was tested with different datasets of multiple scales. Ten of them

[4]For more details about the definition and functioning of MoJoFM and C2C refer to [110, 111, 112]

were open-source code projects which their corresponding MDG was generated using software tools such as *Understand*[5], or NDepend [6]. Besides, they tested FCA with more complex systems such as Firefox web browser and Chromium, as well as an Image Segmentation tool (ITK).

[5]`https://www.scitools.com/`
[6]`https://www.ndepend.com/`

3 Hybrid-Encoded Genetic Algorithm

The proposed genetic algorithm is called Hybrid Encoded Genetic Algorithm (HEGA), as it is based on a hybrid representation of solutions formed by a binary string as well as an integer vector; and the variation operators were selected based on the hybrid nature of this encoding scheme. This shall be discussed in the following subsections.

3.1 Hybrid encoding scheme

The proposal of this thesis begins with the solution encoding. This representation is formed by 2 parts, a binary string and a vector of integer values. The binary part of the encoding represents the number associated to the modules in the graph, while the integer part stands for the total number of modules inside each cluster. Each number in the vector will take the first n modules in the binary string. For instance, take as an example the graph shown in section 2.1.2 now seen in Figure 3.1. An encoded solution will take the numbers of the modules $\{x|x \in [1,8]\}$ and translate them into a binary string, the encoding process must make sure that the bits used for representing the first part of the scheme use the minimum number of bits possible. In this case, the highest number associated with a module is 8 whose binary value is *1000.* The binary string length is four, meaning that the numbers assigned to the modules will be represented with 4 bits.

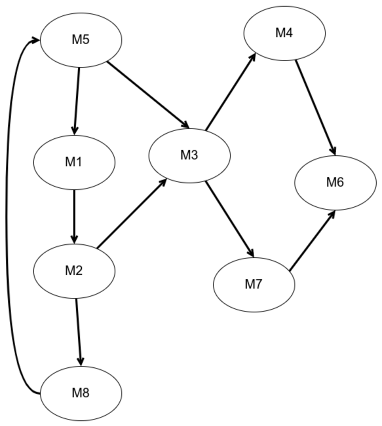


Figure 3.1: MDG example

```
00010011010100101000011101100100
```

3	2	3

Figure 3.2: Encoding example for MDG

The algorithm must ensure that the values inside the vector sum exactly the number of modules in the graph; no more and no less. If any of these constraints fail a repairing process shall be needed. This is represented in Figure 3.2.

3.1.1 Solution decoding

For the solution decoding, the inverse process shall be executed, as can be seen in Figure 3.3, the binary part is decoded by separate. The algorithm should know the length of the highest number of identification of the modules. This can be taken by summing up the values in the integer vector since the modules are named progressively from 1 to n with n being the last module to be counted. Once having the maximum value and, thus, the m bits necessary to represent it, then the binary string is divided into k parts, each of length m. After that the binary representation is translated into a string of numbers. Finally, with the values inside the vector the k numbers are grouped and the partition is obtained as can be seen in Figure 3.4.

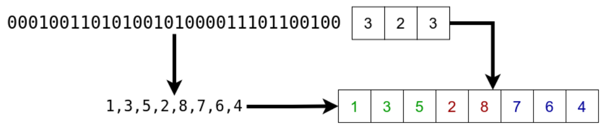


Figure 3.3: Decoding process for an encoded solution

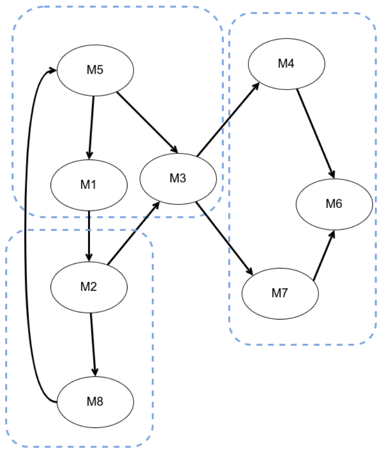


Figure 3.4: Decoded partition for the MDG

3.2 Fitness function

The fitness function is based on the ECA multiobjective approach but integrated into a single-objective multi-factor fitness function. As seen in section 2.4.4, ECA approach aims to optimize the modularization quality, as well as the number of clusters, the cohesion, the coupling, and reduce as much as possible the difference between the biggest cluster and the smallest one. That is why in the first part of the function the *TurboMQ* value is evaluated, then the number of clusters is evaluated in terms of the total number of modules in the graph; finally, the difference between the biggest and the smallest cluster is taken as a negative value since the algorithm is trying to minimize this part. A control value named θ is implemented. This value, obtained empirically, regulates the fitness score of each solution to focus the importance level in the modularization while maintaining the effects of the other factors whithin the function. The fitness value can be obtained by Equations 3.1 – 3.4, where:

- n_{clus} represents the number of clusters in the current partition
- n_{dep} represents the total number of modules in the system

- $|clus_{max}|$ and $|clus_{min}|$ represent the size of the biggest and the smallest cluster respectively

$$f(mdg) = TurboMQ(mdg)\theta + \left(\frac{1-\theta}{2}\right)(NC - \Delta_{clus}) \tag{3.1}$$

$$NC = \frac{n_{clus}}{n_{dep}} \tag{3.2}$$

$$\Delta_{clus} = \frac{|clus_{max}| - |clus_{min}|}{n_{dep}} \tag{3.3}$$

$$\theta = \begin{cases} 1 - \left(\frac{1}{\log_2\left(n_{dep}\right)}\right)^2 & if\ n_{dep} < 10000 \\ 0.9 & otherwise \end{cases} \tag{3.4}$$

3.3 Variation operators

3.3.1 Parent selection

In section 2.3.2 two of the most commonly used selection mechanisms were described. HEGA implements the deterministic binary tournament selection for choosing the parents that are more likely to generate better solutions for the next generation. The reason behind this decision is that the characteristics of the solutions that have a higher fitness value, such as the number of clusters and the distribution of modules inside those clusters, are more likely to be present in solutions with a higher potential than in those with a low MQ value. A deterministic tournament always chooses the best from the n solutions selected. In order to enforce the search in a more efficient area, the best solutions should be selected. Algorithm 4 shows the implementation of the binary tournament used in HEGA.

Algorithm 4 Binary Deterministic Tournament Selection

```
Require: Population $Pop$
Output: A list $Parent_{pool}$ with the best potential solutions for crossover
1: $Parent_{pool}$
2: while $size(Parent_{pool}) < size(Pop)$ do
3:     Randomly choose 2 elements from the population
4:     $\alpha \leftarrow$ the element with the highest fitness value
5:     $Parent_{pool} \leftarrow Parent_{pool} \bigcup \{\alpha\}$        ▷ Add the element to the parent list
```

3.3.2 Crossover operators

The central part of any genetic algorithm is the information exchange to create new individuals. HEGA is no different in that sense, for the recombination a single conventional operator could not be used due to the hybrid nature of the encoding of the solutions. For this reason, two crossover operators were used; one for the binary string and another for the integer vector.

Binary Multivariate Crossover

For the binary string, the Multivariate Crossover Operator (MC) was selected, this operator divides the parent string into n substrings. Then a random value γ is selected for each substring. For each substring where the condition $\gamma < P_c$ is met, a 1-Point crossover mechanism is implemented, otherwise, the substrings are copied to the offspring without alterations. In HEGA, a modification to this operator is implemented, since the binary string represents consecutive numbers, the size of the substrings is the same, and the 1-Point crossover was replaced by an exchange of the full substring. This means that if $\gamma < P_c$ then the substring from the first parent is passed to the second child otherwise it is copied into the first child.

1-Point crossover

The integer vector was also considered for the recombination, for this part of the solution a simple 1-Point crossover operator was used. This is one of the most popular mechanisms used in genetic algorithms. It consists in fragmenting the parents in a single random point and combining the information at that point into the offspring.

Given two parents α_1 and α_2, the first step to perform the 1-Point crossover is to select a random point p_i in which $0 < i \leq n$ where n is the last position in the selected parent. Then the information of α_1 is copied into the first child up to the selected position, then the information in α_2 from p_i and on is copied into the same child. For generating a second offspring this process is repeated but with the parents in exchanged order.

3.3.3 Repair mechanism

One of the most frecuent problems in the execution of the experiment was the presence of invalid solutions. Either the binary string stored duplicated numbers or higher values than the number of modules, or the values inside the vector sum to have more modules than the ones originally placed in the graph. Due to this, it was necessary to implement a repair mechanism when the mutation process was completed.

For the binary string, the values were translated into integer numbers and those positions that stored repeated or outranged values were replaced with the missing values in random order. This can be seen in Figure 3.5. For the vector, the number of the highest cluster will be iteratively reduced by 1 until the number of modules equals the total in the graph as shown in Figure 3.6.

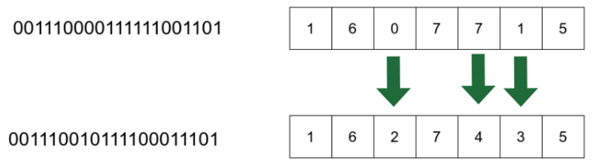


Figure 3.5: Repair mechanism for the binary part

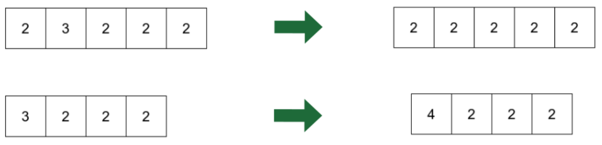


Figure 3.6: Repair mechanism for the integer vector part

3.3.4 Mutation operator

The mutation operator used for this work is not a conventional one. A single conventional operator alone could not help the algorithm to search for better solutions in the search space. It was noticed during the experimentation that, with a single mutation operator either for the integer vector or the binary string, the fitness values were not improved. Even a mutation operator of each part of the representation used at the same time did not improved considerably the results. Furthermore, it became increasingly probable to obtain redundant solutions in the population. To solve this, a mutation mechanism was implemented based on the FICABOO framework [39, 112]. This framework was initially proposed by Zhihua and Tzerpos to be used as a complement to any generic clustering algorithm once the process was complete. As mentioned in Section 2.1.2, it is advisable to deal with omnipresent modules in the clustering process rather than removing them or creating a whole new cluster. Nonetheless, the recent literature shows that recently proposed algorithms either fail to provide a mechanism to cluster omnipresent modules or removes them completely. The proposals that have a mechanism suitable for omnipresent modules apply them after the clustering process is complete. Therefore, it was decided to include an omnipresent module clustering mechanism as part of the evolutionary process. The mutation mechanism proposed is described below.

The first step is to extract the omnipresent values from the Module Dependency Graph. The next step is to run the clustering algorithm selected. Finally, the omnipresent modules are reinserted in the resultant partition using the Orphan Adoption Mechanism (OAM) [47]. This mechanism was proposed by Tzerpos and Holt in 1997 as a clustering tool for inserting new modules into a system or reorganizing its structure.

An orphan is a newly introduced module that is yet to be classified into a cluster. The orphan adoption algorithm evaluates the dependency between the different clusters in a system and the modules to be inserted. Once this process is complete, the algorithm places the new module in the cluster that depends more on it or vice versa. Tzerpos and Holt determined that there are two possible processes in which the orphan adoption mechanism can be applied. The first one is called *incremental clustering*, an orphan adoption applied to this type of module will insert it in an already existing cluster avoiding diminishing the modularization quality. The second process to apply an orphan adoption mechanism is when a structural change is needed in the system, either for a newer version of the system or for redistributing the current one. This process will most certainly need to move a module from one cluster to another one, similar to kidnapping it from its parent subsystem, and hence making it an orphan The process of readopting this kidnapped module into a new cluster is called *corrective clustering*.

HEGA implements a mutation mechanism that is based on FICABOO and the Orphan adoption mechanism. Once the crossover process is finished and the offspring is generated and correctly repaired, the mutation mechanism identifies the global modules within the current MDG partition. These modules are identified according to the definition proposed by Huang and Liu [81]. They say that a module m is considered global if it has a dependency in at least two different clusters other than the one that m belongs to, and either no module is called by m or m is not called by any other module. These modules are obtained for each partition (each solution in the population).

Once the omnipresent modules are identified, the dependency to all the clusters inside the partition is calculated by obtaining a dedication score which is a normalization of the score proposed by Kobayashi et al [44] which evaluates the number of modules inside the same cluster that depend on another module. This can be obtained with Equation 3.5; where $fanin(B)$ represents the number of incoming edges from cluster A to module B, and n_{dep} is the total number of modules inside the system.

$$D(A, B) = \frac{fanin(B)}{n_{dep}} \tag{3.5}$$

Once this value is obtained for all of the omnipresent modules detected, each one should be inserted in the cluster that has the highest potential to improve the general MQ value. This could be determined by evaluating the *TurboMQ* value obtained from inserting each omnipresent module in each cluster. However, this process is computationally expensive when a system is partitioned into a large number of clusters. That is why it was determined that only the clusters with the highest dedication score value obtained for each module would be used. Once these Γ clusters are selected, each module is evaluated and inserted in the cluster that increases the most the modularization value. It can be the case that the general modularization quality is not improved but rather diminished by this process, if that happens then the omnipresent module is returned to its cluster of origin.

Finally, once all the omnipresent modules are evaluated and reinserted in the partition a final MQ evaluation is made. If the value of the modularization of the new partition is better than the original, then the mutation is performed; otherwise, the solution is left unchanged in the population. This last part of the mechanism can lead to the solution not being modified at all despite entering the mutation process. However, this is only the case when the omnipresent modules are located already in the best possible cluster for the general MQ value. After some experiments were performed, it was noticed that 75% of the individuals selected for mutation process were indeed changed for a system that has 13 modules only, and it grows as the numbers of modules and clusters increase. A further description on this particular testing is detailed in section 4.4. The last system that was tested with HEGA had almost 100 modules and 90% of individuals selected for mutation were changed, which means that as the number of modules increases so does the probability of requiring corrective clustering to obtain better solutions. The mutation mechanism used can be seen in Algorithm 5.

3.4 Methodology

The parameters used for the algorithm are based on the ones used for Draco Clustering Tool [16], and proposed in [111]. The population's size and the stochastic behavior of the algorithm are directly related to the size of the MDG. However, a modification was made to improve the results obtained for this algorithm. The parameters were calibrated empirically and later with iRACE. The values that showed the best results were selected to be part of the algorithm and are presented in Figure 3.7. Where MG is the maximum number of generations, PS the number of solutions per each generation, CP the crossover probability, and MP the mutation probability.

In the first stage of the research, the experimentation was based on finding the most suitable variation operators for the algorithm. Five different crossover operators for the

Algorithm 5 Omnipresent modules corrective clustering algorithm

```
Require: A partitioned graph
Output: A new partition with omnipresent modules reorganized
 1: Obtain $MQ$ of the original partition
 2: Extract the omnipresent and global modules of the partition and store them in a
    pool $GP$
 3: for each $mod \in GP$ do
 4:     Calculate the dedication score $DS$ with each cluster
 5:     Select the $\Gamma$ clusters with the highest $DS$
 6:     for each $\gamma_i \in \Gamma$ do
 7:         Calculate TurboMQ with and without $mod \in \gamma_i$
 8:         Calculate $\Delta_{MQ}$
 9:     Insert $mod$ in the cluster with the highest $\Delta_{MQ}$
10:     if No cluster increases the MQ value (All $\Delta$'s are negative) then
11:         Insert the module in the cluster of origin
12: Evaluate $MQ'$ of the resultant partition
13: if $MQ' > MQ$ then
14:     The individual is replaced with the new partition
15: else
16:     The new partition is rejected
```

binary part were tested alongside three different operators for the integer vector used. The binary crossover was tested with: a) Count-Preserving Crosover (CPX), b) One Point Crossover (1PX), c) Uniform Crossover (UX), d) Multivariate Crossover (MC) and e) Random Respectful Crossover (RRC). These operators were taken from Umbarkar and Seth [59]. The integer crossover process was tested with a) One Point Crossover, b) Exchanging the integer vectors on to the offspring, and c) Reversing the order of the values on to the offspring. To determine the best combination of operators, each binary operator was tested with each integer operator. The algorithm was executed 31 times for each combination which means a total amount of 465 experiments. Initially, the parameters were calibrated with different values obtained from [9, 19]. MG and PS were tested with three different values each, CP and MP were tested with other 2 possibilities. Overall, a total amount of 4650 experiments were executed.

After experimenting with different combinations of the operators and values for the parameters, the Multivariate Crossover and One Point Crossover operators were chosen for the recombination of the binary string and the integer vector respectively. These

$$MG = \begin{cases} 50n & n \leq 300 \\ 20n & 300 < n \leq 3000 \\ 5n & 3000 < n \leq 10000 \\ n & n > 10000 \end{cases}$$

$$PS = \begin{cases} 2(n+1) & x \leq 50 \\ n & 50 < x \leq 300 \\ n/2 & 300 < x \leq 1000 \\ n/4 & x > 1000 \end{cases}$$

$$CP = \begin{cases} 0.78 & n \leq 100 \\ 0.78 + 0.2(n-100)/899 & 100 < n < 1000 \\ 1 & otherwise \end{cases}$$

$$MP = \frac{16}{72 \log_2(n)}$$

Figure 3.7: Parameters determined for testing HEGA

were described in section 3.3.2.

For this research, three tools were used to generate the ground truth, Bunch [9], Draco Clustering Tool (DCT) [16], and Fast Clustering Algorithm (FCA) [19]. The first two are stochastic tools since both implement a genetic algorithm. On the other hand, FCA is a deterministic tool, since it obtains the same solution for the same input. This tool in particular is going to be used as a reference only, but the main comparison and results shall be considered with Bunch and Draco. These tools have been tested with a wide range of systems and have obtained good results. That is why they were chosen to be compared with the results obtained by HEGA; Bunch as one of the initial proposals for solving this problem, and Draco as one of the most recent tools.

The algorithm was coded in Python, using Anaconda environment and different libraries to improve performance and cache managing. The algorithm was tested with eight different systems (see Table 3.1), using their corresponding MDG. These were taken from the repositories available for building the same tools that were used as ground truth for HEGA. The systems have different number of modules and dependencies. Also, the largest MDGs used have numerical values associated with each edge

Name	**Nodes**	**Dependencies**
Compiler	13	32
Mtunis	20	57
Ispell	24	103
RCS	29	163
Bison	37	179
Icecast	60	650
Gnupg	88	601
Inn	90	624

Table 3.1: Systems used for testing HEGA

between the nodes of the graph. This refers to the relevance of the relationship between two modules in the system. Finally, the experiments were executed on a personal computer with an Intel®Core i7-10750H CPU, 2.60GHz frequency and 16GB Memory.

4
Results and discussion

The experiments are performed for 31 executions of each stochastic algorithm to be compared (Bunch, DCT and HEGA). Additionally, a comparison with the execution of the FCA tool is documented as a reference with respect to a deterministic process. The executions within the median of the fitness function are used to generate the convergence plots (see Figs. 4.5 - 4.12). The fitness function and the values it is formed by, are plotted independently, and the results of all algorithms are compared. The results are shown below as follows: First the quality of the modularization obtained by HEGA is compared with the other algorithms. Then, the secondary factors that were evaluated within the fitness function (N_{clus} and Δ_{clus}) are also analyzed. Later, the convergence graphs are presented. Finally, some last analysis on the mutation process is explained.

4.1 MQ values comparison

The first and most important value to consider is the quality of the modularization that is obtained after executing the clustering process. In this case, for each system in the dataset the best result, the average modularization value and the standard deviation are documented in Table 4.1. On the other hand, Figure 4.1 shows a more graphical description of the behavior of the algorithm during its best execution. It can be noted that HEGA obtains a better result compared to the other tools. It reaches either the same result or a better one. The only exception to this rule in this experiment is the system *Bison*. The latter is a constant in all the experiments performed. Table

4.1 shows that both, the best result and the average behavior by HEGA were better than Draco's and Bunch's. The results, as seen in the standard deviation, are close to each other. This and the results obtained from the Kurskal-Wallis test, which shall be discussed below, indicate that HEGA is a robust algorithm. And the approach leads to significantly better solutions.

System	**Bunch**			**DCT**			**HEGA**		
	Best MQ	Avg	Std	Best MQ	Avg	Std	Best MQ	Avg	Std
Compiler	1.5064	1.45037	0.024975	1.506494	1.371443	0.092085	**1.506494**	**1.475749**	0.030765
Mtunis	1.7548	1.22414	0.126371	2.286561	2.042981	0.162533	**2.314461**	**2.219911**	0.059808
Ispell	2.0320	1.34070	0.418088	2.267942	2.020409	0.145860	**2.275870**	**2.127648**	0.098751
RCS	1.7116	1.18121	0.079552	2.180971	1.933686	0.157196	**2.199284**	**2.009264**	0.120137
Bison	1.3256	1.19570	0.079552	**2.56962**	**2.25499**	0.155455	2.3198760	2.0964519	0.159823
Icecast	1.5655	1.06934	0.147194	2.448062	2.247255	0.130605	**2.493417**	2.1861290	0.185231
Gnupg	1.8964	1.32461	0.174920	4.166358	3.854349	0.377434	**4.166358**	**3.915400**	0.251278
Inn	2.5364	1.04359	0.090239	5.499570	3.873823	0.510533	**5.505172**	**4.523417**	0.140021

Table 4.1: Best results obtained in 31 executions of HEGA compared with 31 executions of the two stochastic tools used: Bunch and Draco Clustering Tool

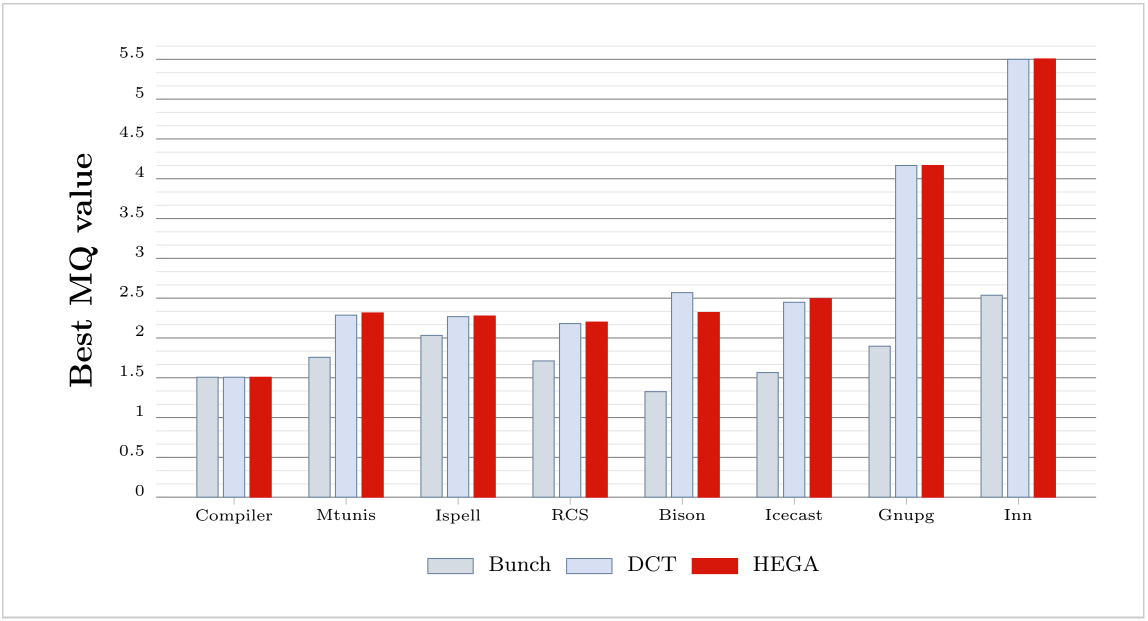
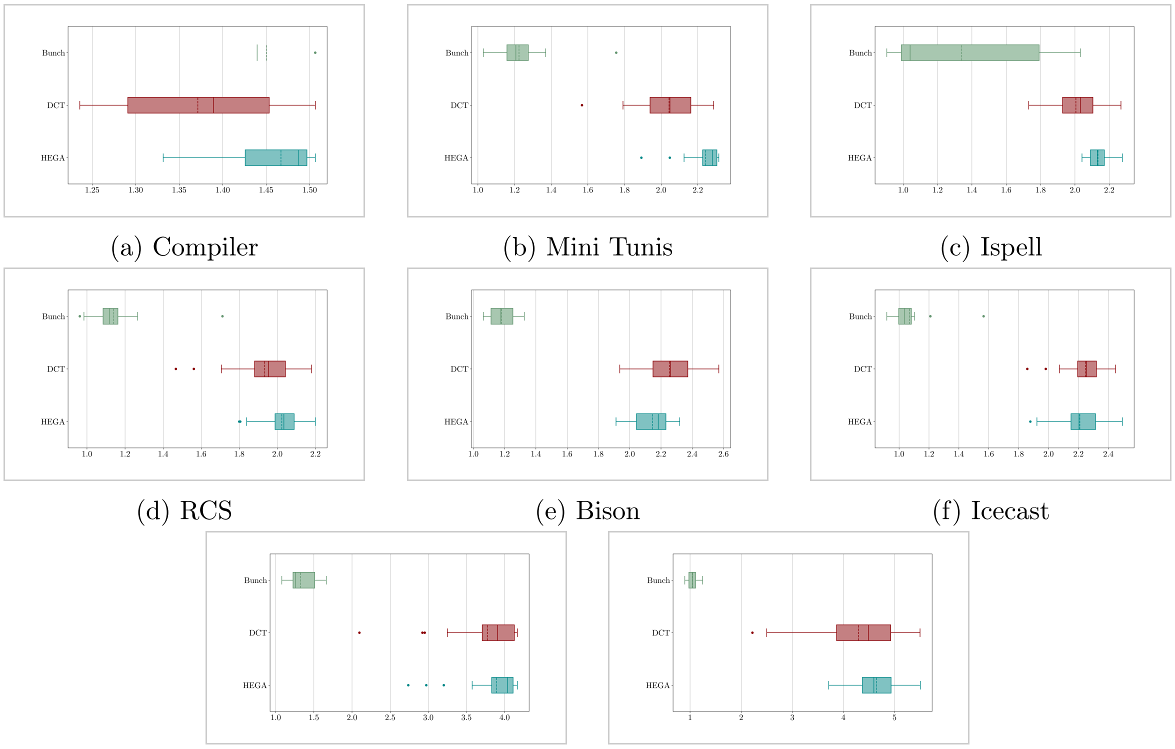


Figure 4.1: Best MQ values obtained by HEGA, Bunch and Draco (The higher, the better)

Now the Kurksal-Wallis and the Bonferroni post-hoc tests are applied to the results. This specific non-parametric tests were chosen beacuse three different samples of results are to be compared. The Kruskal-Wallis test is performed with a significance level $\alpha = 0.05$. Figure 4.2 shows that that there is a significant difference in the dependent variable between the groups obtained from Bunch and HEGA. The confidence intervals obtained for DCT and HEGA show that the groups obtained by DCT do not have mean ranks significantly different from HEGA. However, more robustness in the general performance of the algorithm can be seen in the behavior of HEGA compared to DCT. The results of the statistical test indicate a competitive performance by HEGA against other algorithms in the literature for clustering software systems, by obtaining modularization quality values in a narrower range.



(a) Compiler (b) Mini Tunis (c) Ispell

(d) RCS (e) Bison (f) Icecast

(g) Gnupg (h) Inn

Figure 4.2: Comparison of the general behavior of HEGA, Draco and Bunch according to the Kruskal-Wallis test after 31 executions of each algorithm

As part of the experiments, HEGA was evaluated against a deterministic algorithm, in this case FCA [19]. The results are shown in table 4.2. The table shows the comparison between the best result obtained by HEGA and the only result FCA gives. Due to the stochastic nature of HEGA, it is not possible to evaluate the statistical difference between its results and the results obtained from FCA. Nonetheless, it can be seen that HEGA presents competitive results when compared against a deterministic process. Acording to the literature, FCA proposes a clustering process that only takes into consideration whether there is a relation between the clusters or not. It does not consider if there is a weight associated. HEGA does consider this feature and presents, in some cases, better results than FCA.

System	FCA	HEGA
Compiler	1.494950	**1.506494**
Mtunis	**2.45714**	2.3144610
Ispell	2.043700	**2.275870**
RCS	1.953801	**2.199284**
Bison	**2.49790**	2.3198760
Icecast	1.791809	**2.493417**
Gnupg	3.126215	**4.166358**
Inn	**5.81838**	5.505172

Table 4.2: Best results obtained in 31 executions of HEGA compared with the stochastic tool: Fast Clustering Algorithm

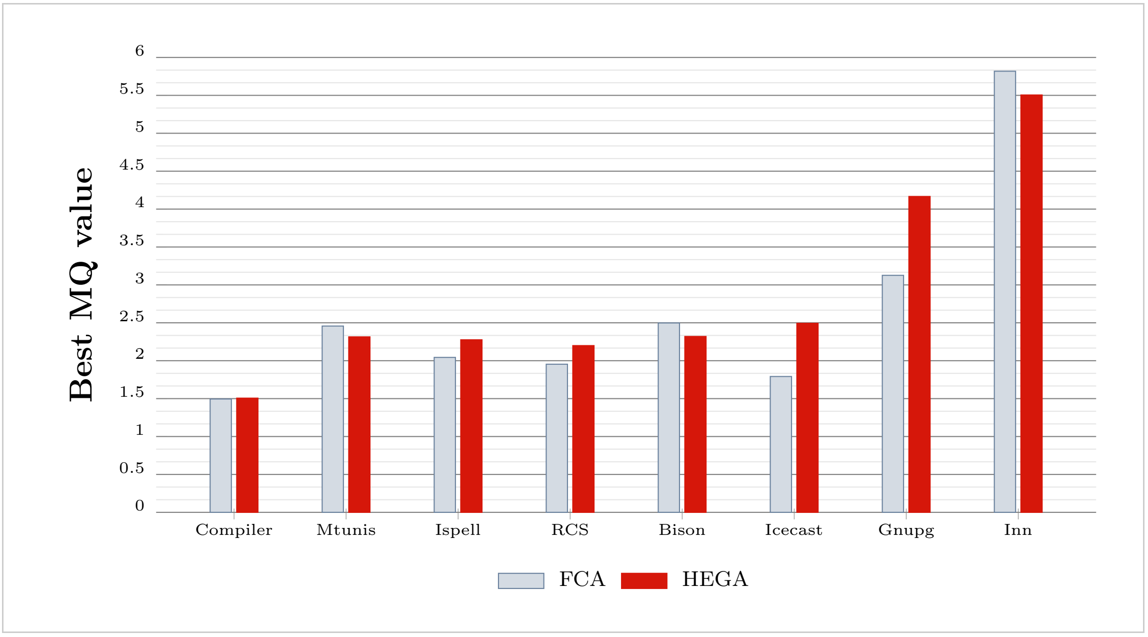


Figure 4.3: Best MQ values obtained by HEGA compared to FCA (The higher, the better)

4.2 Secondary factors comparison

As mentioned in Section 3.2, the solutions were evaluated with a single-objective multi-factor fitness function. The quality of the modularization was already documented in the previous section. The MQ Value has the highest importance, however, the number of clusters and the Δ_{clus} value are also relevant for the approach in which the algorithm is inspired (ECA, see Section 2.4.4). It is worth mentioning that, even though HEGA is not initially proposed as a Multi-Objective approach, it does considers multiple factors to form a single objective function. That is why it becomes necessary to evaluate each of these factors and compare them among the algorithms to observe their impact on the behavior of the clustering process.

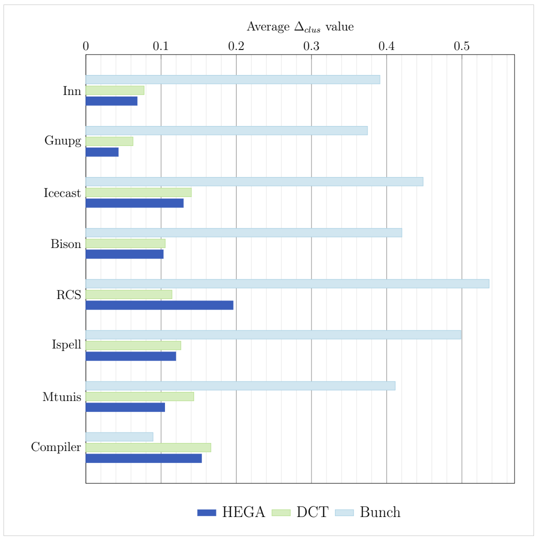
System	**Bunch**			**DCT**			**HEGA**		
	$clus_{max}$	$clus_{min}$	Δ_{clus}	$clus_{max}$	$clus_{min}$	Δ_{clus}	$clus_{max}$	$clus_{min}$	Δ_{clus}
Compiler	3.16	2	**0.0893**	4.13	1.97	0.16625	4.16	2.16	0.15384
Mtunis	13.32	5.10	0.41129	4.61	1.74	0.14354	4.45	2.35	**0.10483**
Ispell	13.42	1.45	0.49865	4.87	1.84	0.12634	4.90	2.03	**0.11962**
RCS	16.68	1.13	0.53615	4.97	1.65	**0.1145**	7.35	1.67	0.195773
Bison	16.68	1.13	0.42022	5.35	1.45	0.10549	6.04	2.23	**0.10297**
Icecast	27.94	1.03	0.44838	9.87	1.45	0.14032	10.23	2.45	**0.12967**
Gnupg	33.95	1.00	0.37445	6.58	1.06	0.06268	7.01	3.21	**0.04318**
Inn	36.25	1.06	0.39103	8.16	1.19	0.07741	9.22	3.08	**0.06819**

Table 4.3: Average size of the biggest cluster ($clus_{max}$), the smallest ($clus_{min}$) and their Δ value after 31 executions of HEGA, Bunch and DCT.

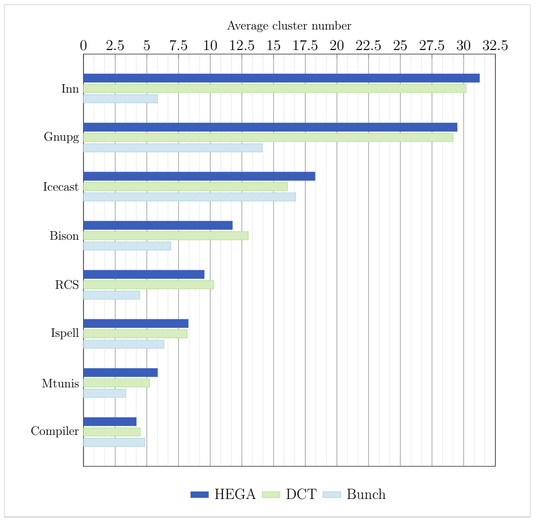
System	**Bunch**	**DCT**	**HEGA**
Compiler	**4.84**	4.48	4.16
Mtunis	3.35	5.21	**5.84**
Ispell	6.35	8.19	**8.26**
RCS	4.45	**10.29**	9.52
Bison	6.90	**13.00**	11.75
Icecast	16.74	16.10	**18.27**
Gnupg	14.10	29.16	**29.48**
Inn	15.85	30.19	**31.25**

Table 4.4: Average number of clusters obtained after 31 executions by HEGA, Bunch and DCT

Table 4.3 shows the comparison of the average values of the difference between the biggest and the smallest cluster obtained with HEGA, Bunch and DCT. The results show that in most cases HEGA presents a better distribution of the modules, as the difference value Δ_{clus} is smaller. Table 4.4 shows that the average number of clusters is also better, in most cases, for HEGA. This behavior suggests that HEGA presents a competing and solid algorithm for clustering software systems in a way that suits good for maximizing the number of clusters and minimizing the difference between the biggest and the smallest cluster. Finally, Figure 4.4 provides a graphical representation of the average values obtained by each algorithm and shows the comparison among them.



(a) Difference value Δ_{clus} between the biggest cluster and the smallest one (the lower the value, the better the modularization)

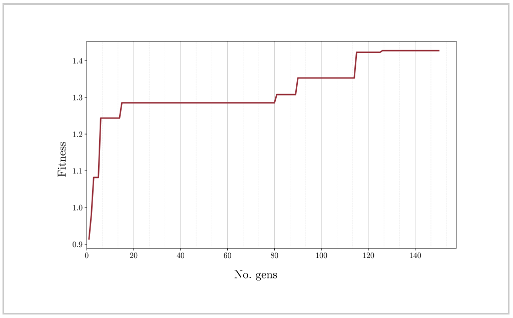


(b) Number of clusters obtained (the higher the value, the better the modularization)

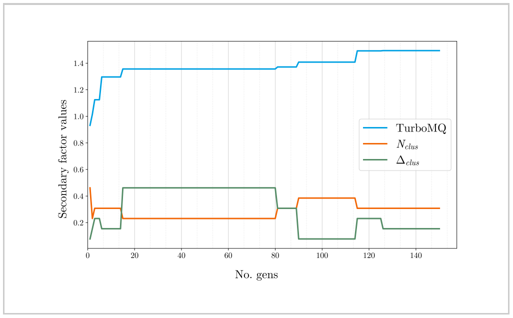
Figure 4.4: Average results of the secondary factors obtained after 31 executions by HEGA, Draco and Bunch

4.3 Convergence analysis

One of the main parts of the analysis of any genetic algorithm is the behavior shown by the convergence of the results. Figures 4.5 - 4.12 show two subfigures each. The first subfigure shows the convergence of the execution in the median of the fitness value, while the second one shows the behavior, during that same execution, of the factors

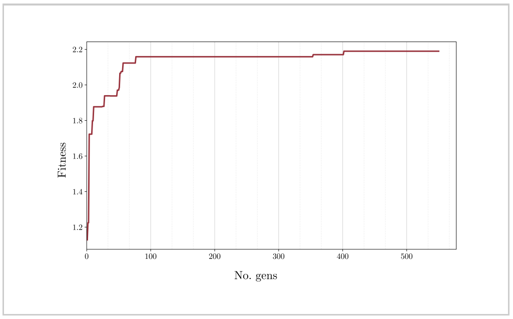


(a)

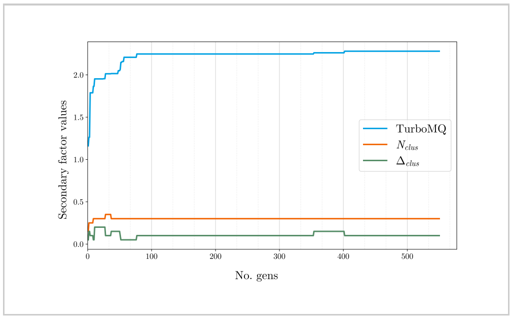


(b)

Figure 4.5: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **Compiler**



(a)

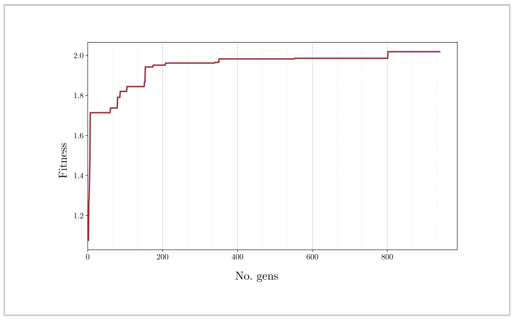


(b)

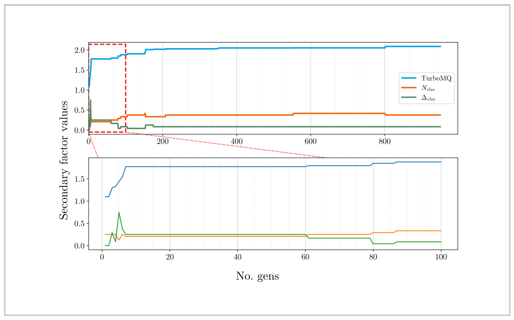
Figure 4.6: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **Mini Tunis**

that compose the fitness function (see Section 3.2). The graphs point out that HEGA has a similar behavior with all the systems. It is made clear that the fitness final score improves as the values of the factors are modified. The variations in the fitness are the result of the control value θ applied to the total sum (see Section 3.2).

There are some cases in which the variations of the factors are difficult to notice since they happen in a relatively fast period of time amongst generations. In those cases figures like Figure 4.8 provide a zoom effect on the interest areas where most of the variations occur. The average convergence speed of HEGA shows that not only the fitness value, but also the quality of the modularization is improved with small variations in the number of clusters and the Δ_{clus} value.

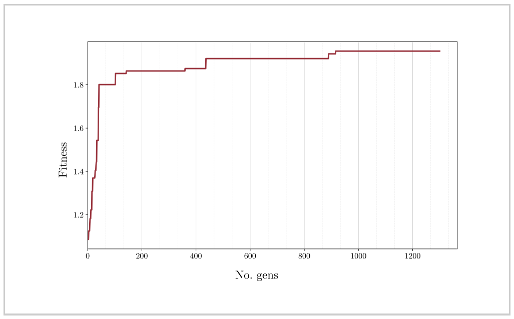


(a)

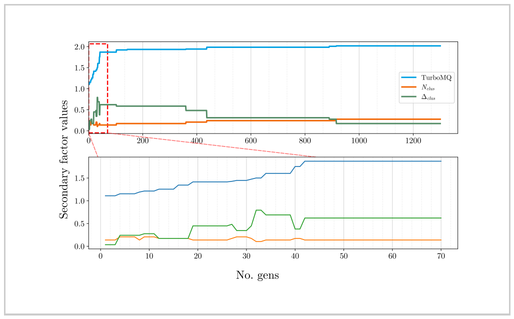


(b)

Figure 4.7: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **Ispell**

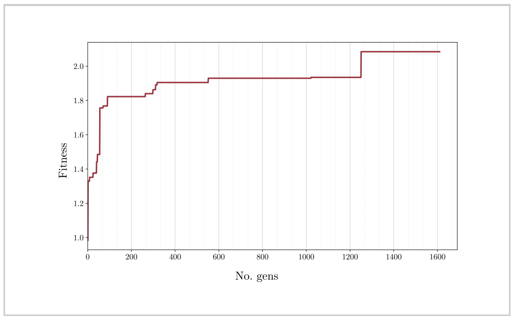


(a)

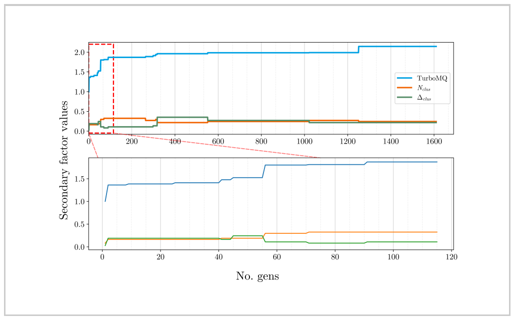


(b)

Figure 4.8: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **RCS**

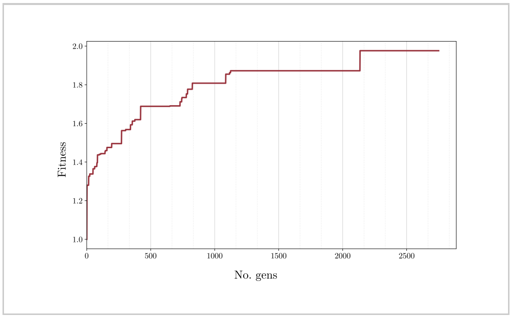


(a)

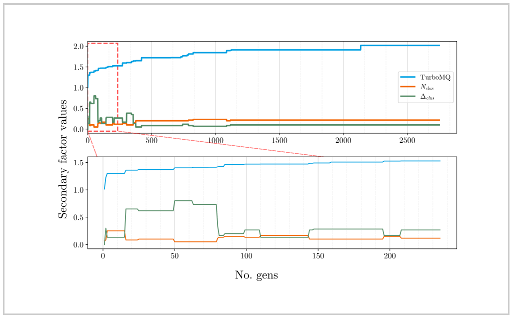


(b)

Figure 4.9: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **Bison**

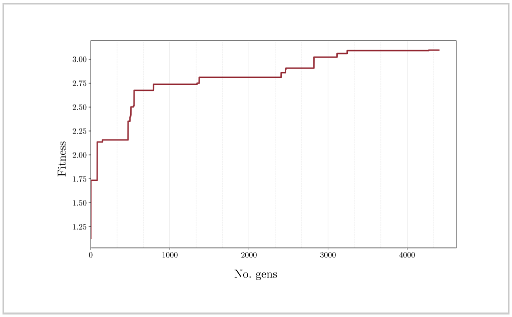


(a)

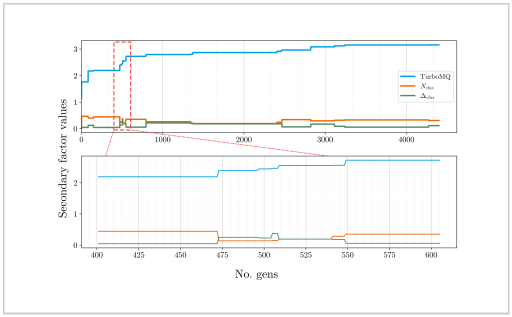


(b)

Figure 4.10: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **Icecast**

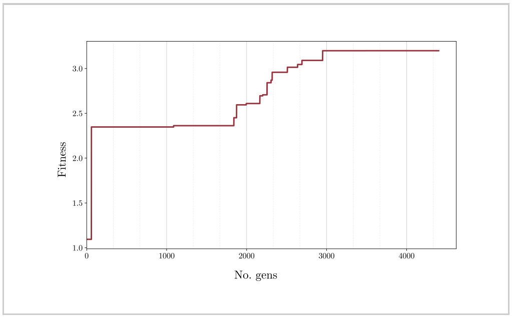


(a)

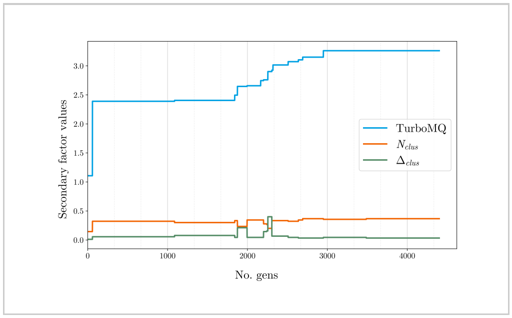


(b)

Figure 4.11: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **Gnupg**



(a)

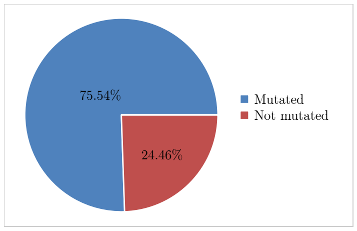


(b)

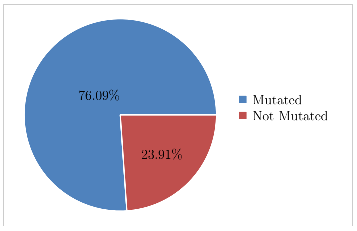
Figure 4.12: Convergence graph of the fitness value (a), and the factors that are evaluated in the fitness function (b) refering to the median execution of HEGA for **Inn**

4.4 Omnipresent modules corrective clustering algorithm evaluation

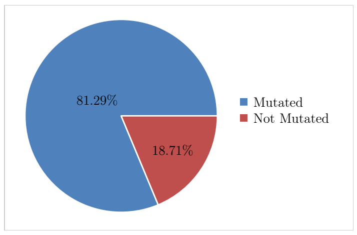
One last experiment was performed in order to evaluate the robustness of HEGA. Due to the unconventional nature of the proposed mutation operator (see Section 3.3.4), a strict analysis was required since the main goal of the mutation process in a genetic algorithm, once the offspring is selected, is to alter the information in the genes of the solution. However, the proposed operator for the mutation process has the probability of rejecting the mutated solution based on the evaluation made after the process. HEGA was executed 31 times to evaluate the performance of the operator. The results shown below refer to the average number of solutions mutated in each generation.



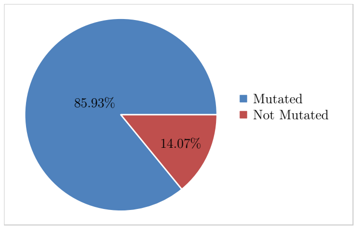
(a) Compiler



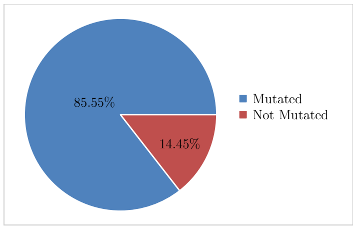
(b) Mtunis



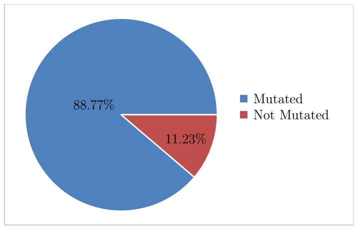
(c) Ispell



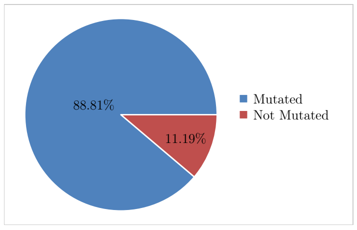
(d) RCS



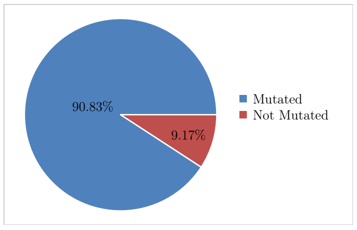
(e) Bison



(f) Icecast



(g) Gnupg



(h) Inn

Figure 4.13: Average percentage of mutated offspring per generation, after 31 executions of HEGA for each of the systems evaluated

Figure 4.13 shows that there is a considerable percentage of the offspring that is not mutated even if they are selected for the process. This percentage tends to decrease as the size of the system (the number of modules to be grouped) increases. For example, Figure 4.13(a) shows that, for the system compiler, almost 25% of the selected solutions are not mutated, whilst for the case of Inn (Figure 4.13(h)) just over 9% of the selected solutions is not altered. This leads the analysis of the experiments to conclude that, for larger systems, the percentage of unaltered selected offspring will be irrelevant or zero.

5
Conclusions

In this thesis, the automatic clustering of software systems has been explored with a genetic algorithm. The main goal established at the beginning of this document was to propose a genetic algorithm that was capable of partitioning Module Dependency Graph (MDG) in such a way that the resultant modularization quality could be improved when compared with the values shown in the speciallized literature. It was documented in Chapter 2 that genetic algorithms have been widely used for solving the Software Module Clustering Problem (SMCP).

To solve the problem stated in Chapters 1 and 2, this document presented the design of a Genetic Algorithm with a Single-Objective Multi-Factored fitness function which evaluates the quality of the modularization alongside the number of clusters and the difference between the biggest and the smallest cluster. These factors are similar to the Multi-Objective approach given to the SMCP which evaluates an Equal-sized Cluster approach. As shown in Chapter 2, modularity is a desired feature in any software system. It refers to the distribution of the components of the constructed product. A good distribution means that the system will be more understandable and, thus, it will be easier to maintain. The problem appears when there is no available documentation and, therefore, the distribution of the components cannot be known. It is necessary to implement a clustering process to determine the architecture of the system based on the relationships among the components of the system.

The proposed algorithm (HEGA) implemented a hybrid encoding of the solutions. The encoding scheme is formed by a binary string which represents the id of every module in the system; and an integer vector that refers to the size of each cluster formed with the numbers in the binary string. Therefore, it was necessary to use two different operators for the crossover process: one for the the binary string and another for the integer vector. HEGA also implemented a mutation operator based on the evaluation of omnipresent modules and corrective clustering, which improved most of the individuals selected for this process. As shown in Section 4.4, the risk of having individuals selected for mutation and not modified gets diminished as the number of clusters and modules grows.

HEGA was tested using eight different systems of different sizes. These systems are part of the most commonly used datasets for testing software clustering tools. After multiple executions of the algorithm proposed in this document, and based on the analysis of the comparison with other clustering tools obtained from the literature, it can be concluded that HEGA presents a similar behavior compared to Draco Clustering Tool, and improved the results shown by Bunch tool. The proposed algorithm reaches to more solid results, as documented in Chapter 4. The latter can be noticed in the standard deviation obtained from the results of the executions. Also the values obtained from the Kruskal-Wallis test show that the values obtained from the modularization between HEGA and Bunch are significantly different. Also; when comparing HEGA and Draco, although the results are not significantly different, their values tend to be more robust, which gives a higher level of trust and more reliability towards HEGA.

As part of the evaluation of HEGA, another comparison was made. This time the average execution of HEGA was compared to the behavior of a deterministic clustering tool called Fast Clustering Algorithm (FCA). The results obtained showed that HEGA presents an acceptable performance despite being a stochastic algorithm. There is an important matter to consider, which refers to FCA not considering omnipresent modules and weighted MDGs. This gives an advantage to HEGA over FCA. Finally, it can be concluded that the hypothesis formulated at the beginning of this document was accepted. HEGA is capable of finding good partitions of Modular Dependency Graphs in such a way that the modules are equally distributed without significantly diminishing the quality of the modularization.

Future work

As further work on this research, more experiments and derived proposals can be implemented. Firstly, an implementation of HEGA in a Multi-Objective scenario is proposed. HEGA, despite having a Single-Objective approach, presents a competitive behavior against MO algorithms such as Draco. Therefore, extending this algorithm into a Multi-Objective approach should follow.

Another objective to be met as further work should be to apply HEGA to bigger systems. Open-source software apps such as mozilla browser tend to be formed by a large number of modules and, thus, can contribute to the evaluation of HEGA.

Finally, further variants of HEGA are encouraged to be implemented, with other operators for crossover and mutation, as well as different selection mechanisms in order to evaluate and improve the robustness and results obtained by the algorithm.

Bibliography

[1] P. Bourque and R. E. Fairley, eds., *Guide to the Software Engineering Body of Knowledge, Version 3.0.* IEEE Computer Society, 2014.

[2] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Computing Surveys*, vol. 45, pp. 1–61, Nov. 2012.

[3] A. Ramirez, J. R. Romero, and C. L. Simons, “A Systematic Review of Interaction in Search-Based Software Engineering,” *IEEE Transactions on Software Engineering*, vol. 45, pp. 760–781, Aug. 2019.

[4] A. H. Farajpour Tabrizi and H. Izadkhah, “Software Modularization by Combining Genetic and Hierarchical Algorithms,” in *2019 5th Conference on Knowledge Based Engineering and Innovation (KBEI)*, (Tehran, Iran), pp. 454–459, IEEE, Feb. 2019.

[5] O. Ramos-Figueroa, M. Quiroz-Castellanos, E. Mezura-Montes, and O. Schütze, “Metaheuristics to solve grouping problems: A review and a case study,” *Swarm and Evolutionary Computation*, vol. 53, p. 100643, Mar. 2020.

[6] L. Araujo and C. Cervigon, *Algoritmos evolutivos: un enfoque practico*. Alfaomega Grupo Editor, S.A. de C.V., 1 ed., 2009. Pages: 332.

[7] A. E. Eiben and J. Smith, “From evolutionary computation to the evolution of things,” *Nature*, vol. 521, pp. 476–482, May 2015.

[8] D. Doval, S. Mancoridis, and B. Mitchell, “Automatic clustering of software systems using a genetic algorithm,” in *STEP ’99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, (Pittsburgh, PA, USA), pp. 73–81, IEEE Comput. Soc, 1999.

[9] S. Mancoridis, B. Mitchell, Y. Chen, and E. 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)*, (Oxford, UK), pp. 50–59, IEEE, 1999.

[10] B. S. Mitchell, *A Heuristic Approach to Solving the Software Clustering Problem.* PhD Thesis, Drexel University, Philadelphia, PA, USA, Mar. 2002.

[11] S. Parsa and O. Bushehrian, “The Design and Implementation of a Framework for Automatic Modularization of Software Systems,” *The Journal of Supercomputing*, vol. 32, pp. 71–94, Apr. 2005.

[12] B. Arasteh, A. Fatolahzadeh, and F. Kiani, “Savalan: Multi objective and homogeneous method for software modules clustering,” *Journal of Software: Evolution and Process*, vol. 34, no. 1, 2022.

[13] D. W. Corne, J. D. Knowles, and M. J. Oates, “The Pareto Envelope-Based Selection Algorithm for Multiobjective Optimization,” in *Parallel Problem Solving from Nature PPSN VI*, vol. 1917, pp. 839–848, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. Series Title: Lecture Notes in Computer Science.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, Apr. 2002.

[15] 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*, vol. 24, pp. 1–45, May 2015.

[16] A. P. M. Tarchetti, L. Amaral, M. C. Oliveira, R. Bonifacio, G. Pinto, and D. Lo, "DCT: An Scalable Multi-Objective Module Clustering Tool," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, (Adelaide, Australia), pp. 171–176, IEEE, Sept. 2020.

[17] K. Praditwong and X. Yao, "A New Multi-objective Evolutionary Optimisation Algorithm: The Two-Archive Algorithm," in *2006 International Conference on Computational Intelligence and Security*, (Guangzhou, China), pp. 286–291, IEEE, Nov. 2006.

[18] F. Morsali and M. R. Keyvanpour, "Search-based software module clustering techniques: A review article," in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, (Tehran), pp. 0977–0983, IEEE, Dec. 2017.

[19] N. Teymourian, H. Izadkhah, and A. Isazadeh, "A fast clustering algorithm for modularization of large-scale software systems," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[20] R. S. Pressman, *Software engineering: a practitioner's approach.* New York: McGraw-Hill Higher Education, 7th ed., 2010.

[21] "ISO/IEC International Standard - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)," *ISO/IEC 25010:2011*, pp. 1–34, 2011.

[22] "ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes," *ISO/IEC/IEEE 12207:2017(E) First edition 2017-11*, pp. 1–157, 2017.

[23] "ISO/IEC/IEEE International Standard - Systems and software engineering– Vocabulary," *ISO/IEC/IEEE 24765:2017(E) Second edition 2017-09*, pp. 1–541, 2017.

[24] S. T. Albin, *The art of software architecture: design methods and techniques.* Indianapolis, Ind: Wiley Pub, 2003.

[25] F. Beck, “Improving Software Clustering with Evolutionary Data,” technical Report, Universität Trier, Feb. 2009.

[26] “IEEE Standard for Information Technology–Systems Design–Software Design Descriptions,” *IEEE STD 1016-2009*, pp. 1–35, 2009.

[27] D. Budgen, *Software design.* Addison-Wesley, 2nd ed., 2003.

[28] R. McLaughlin, “Some notes on program design,” *ACM SIGSOFT Software Engineering Notes*, vol. 16, pp. 53–54, Oct. 1991.

[29] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice.* SEI series in software engineering, Upper Saddle River, NJ: Addison-Wesley, 3rd ed., 2013.

[30] Z. Wang, *Component-based software engineering.* Dissertation, New Jersey Institute of Technology, New Jersey, USA, May 2000.

[31] G. T. Heineman and W. T. Councill, eds., *Component-based software engineering: putting the pieces together.* Boston: Addison-Wesley, 2001.

[32] N. Gupta, A. Rana, and S. Gupta, “Fitness for Solving SMCP Using Evolutionary Algorithm,” *IOP Conference Series: Materials Science and Engineering*, vol. 1099, p. 012041, Mar. 2021.

[33] R. Bazylevych and R. Burtnyk, “Algorithms for software clustering and modularization,” in *2015 Xth International Scientific and Technical Conference "Computer Sciences and Information Technologies" (CSIT)*, (Lviv, Ukraine), pp. 30–33, IEEE, Sept. 2015.

[34] J. Yuste, A. Duarte, and E. G. Pardo, “An efficient heuristic algorithm for software module clustering optimization,” *Journal of Systems and Software*, vol. 190, p. 111349, Aug. 2022.

[35] Q. I. Sarhan, B. S. Ahmed, M. Bures, and K. Z. Zamli, “Software Module Clustering: An In-Depth Literature Analysis,” *IEEE Transactions on Software Engineering*, vol. 48, pp. 1905–1928, June 2022.

[36] O. Räihä, “A survey on search-based software design,” *Computer Science Review*, vol. 4, pp. 203–249, Nov. 2010.

[37] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, “Using automatic clustering to produce high-level system organizations of source code,” in *Proceedings. 6th International Workshop on Program Comprehension. IWPC’98 (Cat. No.98TB100242)*, (Ischia, Italy), pp. 45–52, IEEE Comput. Soc, 1998.

[38] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, “A reverse-engineering approach to subsystem structure identification,” *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[39] Zhihua Wen and V. Tzerpos, “Software Clustering based on Omnipresent Object Detection,” in *13th International Workshop on Program Comprehension (IWPC’05)*, (St. Louis, MO, USA), pp. 269–278, IEEE, 2005.

[40] K. Yano and A. Matsuo, “Moderate Detection and Removal of Omnipresent Modules in Software Clustering,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (Adelaide, Australia), pp. 662–666, IEEE, Sept. 2020.

[41] E. Constantinou, G. Kakarontzas, and I. Stamelos, “An automated approach for noise identification to assist software architecture recovery techniques,” *Journal of Systems and Software*, vol. 107, pp. 142–157, Sept. 2015.

[42] P. Andritsos and V. Tzerpos, “Information-theoretic software clustering,” *IEEE Transactions on Software Engineering*, vol. 31, pp. 150–165, Feb. 2005.

[43] C. Patel, A. Hamou-Lhadj, and J. Rilling, “Software Clustering Using Dynamic Analysis and Static Dependencies,” in *2009 13th European Conference on Software Maintenance and Reengineering*, (Kaiserslautern, Germany), pp. 27–36, IEEE, 2009.

[44] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, “Feature-gathering dependency-based software clustering using Dedication and Modu-

larity," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, (Trento, Italy), pp. 462–471, IEEE, Sept. 2012.

[45] M. Shtern and V. Tzerpos, "Methods for selecting and improving software clustering algorithms: METHODS FOR SELECTING AND IMPROVING SOFTWARE CLUSTERING ALGORITHMS," *Software: Practice and Experience*, vol. 44, pp. 33–46, Jan. 2014.

[46] V. Tzerpos and R. Holt, "ACCD: an algorithm for comprehension-driven clustering," in *Proceedings Seventh Working Conference on Reverse Engineering*, (Brisbane, Qld., Australia), pp. 258–267, IEEE Comput. Soc, 2000.

[47] V. Tzerpos and R. Holt, "The Orphan Adoption problem in architecture maintenance," in *Proceedings of the Fourth Working Conference on Reverse Engineering*, (Amsterdam, Netherlands), pp. 76–82, IEEE Comput. Soc, 1997.

[48] A. M. Turing, "I.— computing machinery and intelligence," *Mind*, vol. LIX, pp. 433–460, Oct. 1950.

[49] J. R. Searle, "Minds, brains, and programs," *Behavioral and Brain Sciences*, vol. 3, no. 3, pp. 417–424, 1980.

[50] D. B. Fogel, *Evolutionary computation: toward a new philosophy of machine intelligence*. Hoboken, N.J.: John Wiley & Sons, 3rd ed ed., 2006. OCLC: 85820773.

[51] J. C. Bezdek, "Computational Intelligence Defined - By Everyone !," in *Computational Intelligence: Soft Computing and Fuzzy-Neuro Integration with Applications* (O. Kaynak, L. A. Zadeh, B. Türkşen, and I. J. Rudas, eds.), pp. 10–37, Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[52] R. C. Eberheart, P. K. Simpson, and R. W. Dobbins, *Computational intelligence PC tools*. Academic Press Professional, Inc., 3 ed., Aug. 1996.

[53] R. Penrose, *The emperor's new mind: concerning computers, minds and the laws of physics*. Oxford: Oxford Univ. Pr, 1999.

[54] J. C. Bezdek, “REPRINT: The History, Philosophy and Development of computational intelligence (How a simple tune became a monster hit),” *Computational Intelligence*, Jan. 1994.

[55] J. C. Bezdek, “On the relationship between neural networks, pattern recognition and intelligence,” *International Journal of Approximate Reasoning*, vol. 6, pp. 85–107, Feb. 1992.

[56] S. Binitha and S. S. Sathya, “A Survey of Bio inspired Optimization Algorithms,” *International Journal of Soft Computing and Engineering*, vol. 2, no. 2, pp. 137–151, 2012.

[57] A. Eiben and J. Smith, *Introduction to Evolutionary Computing.* Springer, 2015.

[58] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* Complex adaptive systems, MIT Press, 1st mit press ed ed., 1992.

[59] A. Umbarkar and P. Seth, “Crossover operators in genetic algorithms: A review,” *ICTACT Journal on Soft Computing*, vol. 06, pp. 1083–1092, Oct. 2015.

[60] S. Ronald, “Robust encodings in genetic algorithms: a survey of encoding issues,” in *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC ’97)*, (Indianapolis, IN, USA), pp. 43–48, IEEE, 1997.

[61] N. H. Siddique and H. Adeli, *Computational intelligence: synergies of fuzzy logic, neural networks, and evolutionary computing.* Chichester, West Sussex, United Kingdom: John Wiley & Sons, 2013.

[62] D. E. Goldberg and K. Deb, “A Comparative Analysis of Selection Schemes Used in Genetic Algorithms,” in *Foundations of Genetic Algorithms*, vol. 1, pp. 69–93, Elsevier, 1991.

[63] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* New York: Oxford University Press, 1996.

[64] A. Brindle, *Genetic algorithms for function optimization.* PhD Thesis, University of Alberta, Edmonton, Alberta; Canada, Aug. 1980. Publisher: University of Alberta Libraries.

[65] K. A. De Jong, "Analysis of the behavior of a class of genetic adaptive systems," Technical report 185, University of Michigan, Ann Arbor, Michigan; USA, Aug. 1975.

[66] L. B. Booker, *Intelligent Behavior as an Adaptation to the Task Environment.* PhD Thesis, University of Michigan, Ann Arbor, Michigan; USA, 1982.

[67] K.-M. Lee, T. Yamakawa, and Keon-Myung Lee, "A genetic algorithm for general machine scheduling problems," in *1998 Second International Conference. Knowledge-Based Intelligent Electronic Systems. Proceedings KES'98 (Cat. No.98EX111)*, vol. 2, (Adelaide, SA, Australia), pp. 60–66, IEEE, 1998.

[68] F. Sivrikaya-Şerifoğlu and G. Ulusoy, "Parallel machine scheduling with earliness and tardiness penalties," *Computers & Operations Research*, vol. 26, pp. 773–787, July 1999.

[69] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," *IEEE Transactions on Software Engineering*, vol. 36, pp. 742–762, Nov. 2010.

[70] C. Sharma, S. Sabharwal, and R. Sibal, "A Survey on Software Testing Techniques using Genetic Algorithm," *IJCSI International Journal of Computer Science Issues*, vol. 10, no. 1, 2013.

[71] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, (London England), pp. 1098–1105, ACM, July 2007.

[72] G. I. Latiu, O. A. Cret, and L. Vacariu, “Automatic Test Data Generation for Software Path Testing Using Evolutionary Algorithms,” in *2012 Third International Conference on Emerging Intelligent Data and Web Technologies*, (Bucharest, Romania), pp. 1–8, IEEE, Sept. 2012.

[73] S. Wappler and F. Lammermann, “Using evolutionary algorithms for the unit testing of object-oriented software,” in *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO ’05*, (Washington DC, USA), p. 1053, ACM Press, 2005.

[74] P. Valenzuela-Toledo, C. Cares, and M. Dieguez, “Search Based Risk Reduction Supporting the Intelligent Components Selection Process,” in *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, (Valparaiso, Chile), pp. 1–7, IEEE, Nov. 2019.

[75] S. A. Saleh, S. K. Ahmed, and F. S. Nashat, “A Genetic Algorithm for Solving an Optimization Problem: Decision Making in Project Management,” in *2020 International Conference on Computer Science and Software Engineering (CSASE)*, (Duhok, Iraq), pp. 221–225, IEEE, Apr. 2020.

[76] T. J. Gandomani, M. Dashti, and M. Z. Nafchi, “Hybrid Genetic-Environmental Adaptation Algorithm to Improve Parameters of COCOMO for Software Cost Estimation,” in *2022 Second International Conference on Distributed Computing and High Performance Computing (DCHPC)*, (Qom, Iran, Islamic Republic of), pp. 82–85, IEEE, Mar. 2022.

[77] S. Ma and S. S. Deng, “Research on Software Project Scheduling Based on Genetic Algorithm,” in *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, (Nanchang, China), pp. 67–70, IEEE, Mar. 2021.

[78] F. Sarro, A. Petrozziello, and M. Harman, “Multi-objective software effort esti-

mation," in *Proceedings of the 38th International Conference on Software Engineering*, (Austin Texas), pp. 619–630, ACM, May 2016.

[79] F. A. Batarseh, R. Mohod, A. Kumar, and J. Bui, "The application of artificial intelligence in software engineering," in *Data Democracy*, pp. 179–232, Elsevier, 2020.

[80] M. Saeed, O. Maqbool, H. Babri, S. Hassan, and S. Sarwar, "Software clustering techniques and the use of combined algorithm," in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, (Benevento, Italy), pp. 301–306, IEEE Comput. Soc, 2003.

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

[82] Chenlong Liu, Jing Liu, and Zhongzhou Jiang, "A Multiobjective Evolutionary Algorithm Based on Similarity for Community Detection From Signed Social Networks," *IEEE Transactions on Cybernetics*, vol. 44, pp. 2274–2287, Dec. 2014.

[83] K. Mahdavi, M. Harman, and R. Hierons, "A multiple hill climbing approach to software module clustering," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, (Amsterdam, Netherlands), pp. 315–324, IEEE Comput. Soc, 2003.

[84] 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)*, (Shiraz), pp. 55–60, IEEE, Oct. 2017.

[85] O. Räihä, E. Mäkinen, and T. Poranen, "Using simulated annealing for producing software architectures," in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, (Montreal Québec Canada), pp. 2131–2136, ACM, July 2009.

[86] K. Jeet and R. Dhir, "Software Module Clustering Using Hybrid SocioEvolutionary Algorithms," *International Journal of Information Engineering and Electronic Business*, vol. 8, pp. 43–53, July 2016.

[87] A. S. Mamaghani and M. Hajizadeh, "Software modularization using the modified firefly algorithm," in *2014 8th. Malaysian Software Engineering Conference (MySEC)*, (Langkawi, Malaysia), pp. 321–324, IEEE, Sept. 2014.

[88] V. Kumar, J. K. Chhabra, and D. Kumar, "Grey Wolf Algorithm-Based Clustering Technique," *Journal of Intelligent Systems*, vol. 26, pp. 153–168, Jan. 2017.

[89] A. Chhabra and J. K. Chhabra, "TA-ABC: Two-Archive Artificial Bee Colony for Multi-objective Software Module Clustering Problem," *Journal of Intelligent Systems*, vol. 27, pp. 619–641, Oct. 2018.

[90] Amarjeet and J. K. Chhabra, "Many-objective artificial bee colony algorithm for large-scale software module clustering problem," *Soft Computing*, vol. 22, pp. 6341–6361, Oct. 2018.

[91] Amarjeet and J. K. Chhabra, "FP-ABC: Fuzzy-Pareto dominance driven artificial bee colony algorithm for many-objective software module clustering," *Computer Languages, Systems & Structures*, vol. 51, pp. 1–21, Jan. 2018.

[92] M. Bishnoi and P. Singh, "Modularizing Software Systems using PSO optimized hierarchical clustering," in *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, (New Delhi, India), pp. 659–664, IEEE, Mar. 2016.

[93] A. Prajapati and J. K. Chhabra, "A Particle Swarm Optimization-Based Heuristic for Software Module Clustering Problem," *Arabian Journal for Science and Engineering*, vol. 43, pp. 7083–7094, Dec. 2018.

[94] A. Prajapati, "A particle swarm optimization approach for large-scale many-objective software architecture recovery," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, pp. 8501–8513, Nov. 2022.

[95] H. Izadkhah and M. Tajgardan, “Information Theoretic Objective Function for Genetic Software Clustering,” in *The 5th International Electronic Conference on Entropy and Its Applications*, p. 18, MDPI, Nov. 2019.

[96] R. Mahouachi, “Search-Based Cost-Effective Software Remodularization,” *Journal of Computer Science and Technology*, vol. 33, pp. 1320–1336, Nov. 2018.

[97] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, and R. Oliveto, “Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization,” in *Search Based Software Engineering*, vol. 7515, pp. 75–89, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[98] C. Schroder, A. van der Feltz, A. Panichella, and M. Aniche, “Search-Based Software Re-Modularization: A Case Study at Adyen,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, (Madrid, ES), pp. 81–90, IEEE, May 2021.

[99] S. Parsa and O. Bushehrian, “A New Encoding Scheme and a Framework to Investigate Genetic Clustering Algorithms,” *Journal of Research and Practice in Information Technology*, vol. 37, no. 1, p. 17, 2005.

[100] B. Khan, S. Sohail, and M. Y. Javed, “Evolution Strategy Based Automated Software Clustering Approach,” in *2008 Advanced Software Engineering and Its Applications*, (Hainan, China), pp. 27–34, IEEE, 2008.

[101] K. Praditwong, “Solving software module clustering problem by evolutionary algorithms,” in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, (Nakhon Pathom, Thailand), pp. 154–159, IEEE, May 2011.

[102] K. Praditwong, M. Harman, and X. Yao, “Software Module Clustering as a Multi-Objective Search Problem,” *IEEE Transactions on Software Engineering*, vol. 37, pp. 264–282, Mar. 2011.

[103] K. Praditwong, M. Harman, and X. Yao, "Software Module Clustering using Metaheuristic Search Techniques: A Survey," *IEEE Transactions on Software Engineering*, vol. 37, pp. 264–282, Mar. 2011.

[104] T. Deepika and R. Brindha, "Multi objective functions for software module clustering with module properties," in *2012 International Conference on Communication and Signal Processing*, (Chennai, India), pp. 17–22, IEEE, Apr. 2012.

[105] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, (Vancouver BC Canada), pp. 1263–1270, ACM, July 2014.

[106] D. W. Corne, J. D. Knowles, and M. J. Oates, "The Pareto Envelope-Based Selection Algorithm for Multiobjective Optimization," in *Parallel Problem Solving from Nature PPSN VI* (G. Goos, J. Hartmanis, J. van Leeuwen, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, eds.), vol. 1917, pp. 839–848, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. Series Title: Lecture Notes in Computer Science.

[107] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using Cohesion and Coupling for Software Remodularization: Is It Enough?," *ACM Transactions on Software Engineering and Methodology*, vol. 25, pp. 1–28, Aug. 2016.

[108] M. Harman, R. Hierons, and M. Proctor, "A new representation and crossover operator for search-based optimization of software modularization," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pp. 1351–1358, 2002.

[109] J. Sun, Y. Xu, and W. Shuyan, "PSO with Reverse Edge for Multi-Objective Software Module Clustering," *International Journal of Performability Engineering*, 2018.

[110] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, “Comparing Software Architecture Recovery Techniques Using Accurate Dependencies,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, (Florence, Italy), pp. 69–78, IEEE, May 2015.

[111] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, “Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques,” *IEEE Transactions on Software Engineering*, vol. 44, pp. 159–181, Feb. 2018.

[112] Zhihua Wen and V. Tzerpos, “An effectiveness measure for software clustering algorithms,” in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, (Bari, Italy), pp. 194–203, IEEE, 2004.