UNIVERSIDAD VERACRUZANA

Artificial Intelligence Research Institute

Efficient Heuristic Strategies for the Parallel-Machine Scheduling Problem with Unrelated Machines and Makespan Minimization

Submitted by:

M.A.I. Octavio Ramos-Figueroa

AS THE FULFILLMENT OF REQUIREMENT FOR THE DEGREE OF

Ph.D. in Artificial Intelligence

Advisor:

Dr. Marcela Quiroz-Castellanos

Co-advisor:

Dr. Efrén Mezura-Montes

XALAPA, VERACRUZ, MÉXICO June, 2022.

Acknowledgments

The research project reported in this thesis would not have been possible without the scientific, personal, and financial support of many people and organizations. Therefore, I would like to thank:

- The Instituto de Investigaciones en Inteligencia Artificial of the Universidad Veracruzana (IIIA-UV) for the support during my P.h.D.
- The Consejo Nacional de Ciencia y Tecnología (CONACYT) for the financial support provided during the completion of this research project.
- My thesis advisor, Dr. Marcela Quiroz Castellanos, for the patience, knowledge, commitment, and trust throughout these four years in the P.h.D., a fundamental part to conclude this work in a timely manner.
- My co-advisor, Dr. Efrén Mezura Montes, for all the contributions to this research work.
- The research group of the IIIA-UV for their attention and pleasant teachings that I will never forget.
- To Dr. Laura Cruz Reyes for the support and attention during my research stay at the Instituto Tecnológico de Ciudad Madero, who shared her knowledge and made great contributions to this work.
- I thank all my colleagues from the Master's and Ph.D. in Artificial Intelligence for sharing their experience and knowledge, but above all their friendship. Thank you for making me feel at home.
- To my friends for their emotional support, advice, and words of encouragement.
- But above all to my family and especially to my mother Maria Amparo Figueroa Guerrero for her unconditional support, who continues to be fundamental support and has given me the best teachings in life.

Abstract

Many problems of practical and theoretical importance within the fields of Artificial Intelligence and Operations Research are combinatorial. Combinatorial optimization problems consist of finding values for discrete variables that meet certain conditions and maximize (or minimize) an objective function. Usually, the problems with these characteristics are easy to define, but often difficult to solve. In such a way that, the solution process of many of these problems represents a great challenge and currently there is no algorithm to find the optimal solution efficiently in the worst case. Such problems have been classified by the scientific community as belonging to the NP-hard class.

This work focuses on NP-hard grouping problems, a special class of combinatorial problems that, in general, implies to search an efficient distribution of a collection of items among a set of groups. Due to the difficulty of this type of problem, the specialized literature contains several algorithms for their solution, mainly metaheuristic algorithms, that can obtain high-quality approximated solutions in short execution times. The most outstanding proposals include local searches, swarm intelligence algorithms, and evolutionary algorithms, highlighting the results obtained by the Grouping Genetic Algorithm (GGA). However, despite the efforts of the scientific community in the development of new strategies, to date, there is no algorithm that presents the best performance for all possible situations. This phenomenon has been in-deep studied and has been described through the No-Free-Lunch theorem, an impossibility theorem establishing that (1) a general-purpose optimization strategy is impossible and (2) the only way one solution method can exceed another is if it uses specific-purpose heuristics designed with knowledge of the problem domain under consideration. For this reason, the development of high-performance algorithms for NP-hard problems is an open research field and specialists throughout the world work on the development of new solution methods. From the specialized literature also emerges that much of the recent progress in algorithm development has been aided by a better understanding of the properties of problem instances and the optimization process of the algorithms that solve them.

The foregoing motivates this research work that addresses the characterization of the problem Parallel Machine-Scheduling with Unrelated Machines and Makespan Minimization ($R||C_{max}$) and the optimization process of the GGA. In this way, this work covers (1) the design of the first GGA for $R||C_{max}$, since, as the best of our knowledge the literature does not includes another proposal with a GGA for this problem; (2) the development of an Enhanced GGA (EGGA) based on a systematical study of the optimization process presented by the heuristic strategies of each GGA component (population initialization, crossover and

mutation operators, and reproduction technique); and (3) the characterization of the $R||C_{max}$ structure and the EGGA algorithmic behavior to identify improvement opportunity niches and design a Final GGA (FGGA).

The systematical study of each GGA component in isolation allowed generating intelligent strategies to improve their performance. Likewise, the characterization approach allowed identifying the properties in the structure of an $R||C_{max}$ instance that becomes it difficult. Finally, it allowed understanding how the $R||C_{max}$ properties impact the optimization process and the final performance of the proposed EGGA. The knowledge gained was used to design the FGGA.

The experimental results showed the usefulness of the characterization approach employed, since the improvements implemented from the initial GGA to the FGGA allowed reaching an improvement rate of about 396%. In such a way that the FGGA exceeds the effectiveness of the state-of-the-art solution methods by using only 10,000 generations.

The final outcome of this work demonstrates the importance of knowing in-depth the problem to solve, since such knowledge can be used to improve the performance of the existing solution algorithms and to design new ones. We expect that the approach shown in this work will be used as a guide for the study of other grouping problems and for the design of high-performance solution methods.

Contributions of the thesis work

- Conference talks:
  - Ramos-Figueroa, O., & Quiroz-Castellanos, M. (2019). Metaheuristics to solve grouping problems: A review. Presented at the 16th Esicup meeting.
  - Ramos-Figueroa, O., & Quiroz-Castellanos, M. (2020). An Experimental Study of Grouping Mutation Operators for the Unrelated Parallel-Machine Scheduling Problem. Presented at the 8th International Workshop on Numerical and Evolutionary Optimization meeting.
  - Ramos-Figueroa, O., & Quiroz-Castellanos, M. (2021, July). A grouping genetic algorithm for the unrelated parallel-machine scheduling problem. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (pp. 135-136).
- Book chapters:
  - Ramos-Figueroa, O., Quiroz-Castellanos, M., Carmona-Arroyo, G., Vázquez, B., & Kharel, R. (2021). Parallel-machine scheduling problem: An experimental study of instances difficulty and algorithms performance. In Recent Advances of Hybrid Intelligent Systems Based on Soft Computing (pp. 13-49). Springer, Cham.
- Journal papers:
  - Ramos-Figueroa, O., Quiroz-Castellanos, M., Mezura-Montes, E., & Schütze, O. (2020). Metaheuristics to solve grouping problems: A review and a case study. Swarm and Evolutionary Computation, 53, 100643. case study. Swarm and Evolutionary Computation, 53, 100643.
  - Ramos-Figueroa, O., Quiroz-Castellanos, M., Mezura-Montes, E., & Kharel, R. (2021). Variation operators for grouping genetic algorithms: A review. Swarm and Evolutionary Computation, 60, 100796.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Many problems of practical and theoretical importance within the fields of Artificial Intelligence and Operations Research are combinatorial. The literature includes Combinatorial Optimization Problems (COPs) in several relevant contexts such as engineering, science, economics, and everyday life. For this reason, several studies have been carried out to produce practical and theoretical knowledge that help to solve these problems efficiently. In general, the COPs involve finding values for discrete variables to obtain an optimal solution, considering certain constraints and conditions. Thus, the algorithms that solve this type of problem seek an arrangement, grouping, order, or selection of discrete objects, usually finite in number, that maximize or minimize an objective function, respecting the given constraints [1].

Frequently, the COPs are easy to define, but often hard to solve. In such a way that the solution process required for many of these problems represents a great challenge, and currently, there is no algorithm to find the optimal solution efficiently for the worst case. Those problems belong to the NP-hard class, and are considered inherently intractable from a computational point of view [2].

The state-of-the-art includes a great variety of solution methods designed to solve NP-hard COPs, highlighting the results obtained by metaheuristics that can reach high-quality approximate solutions in short execution times. However, it is important to note that, despite the efforts of the scientific community in the development of new strategies, to date, no algorithm is the best option for all possible situations [3]. Given the above, the development of high-performance algorithms for NP-hard problems is an open field of research, and specialists throughout the world work on the design of new solution strategies.

This research focuses on a particular class of combinatorial optimization problems, known as grouping problems. Nowadays, solving such problems has become an important issue, mainly because many of them occur in the industry, hospitals, seaports, and many other real-world applications. This work concentrates on the

Parallel-Machine Scheduling with unrelated machines and makespan minimization, also known as $R||C_{max}$, which belongs to the class of grouping problems.

The main motivation to carry out this research is related to the characterization of the problem $R||C_{max}$ and the analysis of the algorithmic behavior of metaheuristic algorithms specially designed to solve this problem. The detail of the $R||C_{max}$ problem will be explained in Chapter 3. In this way, we want to provide useful tools for developing efficient strategies that incorporate knowledge of the problem-domain in grouping problems with different constraints and conditions, using solution methods for $R||C_{max}$ as a guide.

In accordance with the specialized literature, the Grouping Genetic Algorithm (GGA) is one of the most used solution methods to solve this type of problems. The GGA procedure will be explained in Chapter 4. The state-of-the-art highlights the results of GGAs with variation operators that modify the solutions in a controlled way, using different criteria to modify the genetic material of the solutions according to the properties of the problem to solve. This work seeks to demonstrate how the GGA performance can improve by incorporating operators designed using knowledge of the problem-domain. In this way, we expect that the design of GGAs under this approach can be adapted to solve other grouping problems.

1.2 Problem statement

Grouping problems are known for their high difficulty, in such a way that so many of them belong to the NP-hard class. It is important to note that, although the central idea of these problems is to find an efficient distribution of a set of items among a collection of groups D, there are grouping problems with different characteristics. Thus, grouping problems can be *constant* or *variable* according to the number of groups. In a *constant* problem, the number of groups D is known, and the objective is to identify the efficient distribution of the items among the D groups. Therefore, in *variable* problem the number of groups is unknown, and the goal will be to find the most efficient grouping that optimizes the D value. On the other hand, grouping problems can have *identical* and $non-identical$ groups. If the quality of a solution is affected by swapping all items of two groups, that problem belongs to the $non-identical$ grouping class. Otherwise, the problem is part of the *identical* category. Finally, grouping problems can be *order dependent* when the quality of the solution depends on the order of the groups. Consequently, grouping problems without such a dependency belong to the class *no order dependent* [4].

The scientific community has shown a great interest in these problems, mainly, because most of them emerge in real-world problems, like Home Health Care, Facility Location, Economy of Scale, Cell Formation, Material Cutting, and Stock Portfolio, to mention some examples.

Although the specialized literature includes several metaheuristic algorithms for grouping problems, it highlights the popularity of the Grouping Genetic Algorithm (GGA), related to its promising results and its flexibility to adopt new ideas to

handle problems with different constraints and conditions. However, despite all the efforts carried out to develop high performance solution methods, currently, there is no algorithm that shows the best results for all possible situations. Therefore, this is still an open research area.

In this sense, the state-of-the-art suggests that one of the key points for the design of efficient heuristic algorithms is to identify which strategies make an algorithm to show a better performance and under what conditions they obtain it. Recent works remark that much of the recent progress in the development of high performance algorithms has been aided by a better understanding of the properties immersed on the problem instances, the optimization process of the algorithms that solve them, and their final performance. Another important fact is that there are still emerging new and more complex grouping problems. Therefore, it is necessary to generate knowledge of the problem domain that could help in the solution of future problems.

It is important to note that, although the percentage of studies carried out to understand how and why the algorithms follow a particular behavior is low, the specialized literature includes some efforts such as the seminal work of Quiroz-Castellanos that present an approach to characterize the NP-hard grouping problem Bin Packing. The information gained from such study was used to design a high performance GGA using intelligent strategies that incorporates knowledge of the Bin Packing problem domain [5].

The above motivates this research work that looks for demonstrating that it is possible to generalize the use of characterization approach of Quiroz-Castellanos to the design of efficient and robust GGAs for grouping problems with different characteristics. In this sense, this research project addresses the characterization of the NP-hard grouping problem Parallel Machine-Scheduling with Unrelated Machines and Makespan Minimization $R||C_{max}$ since, according to the specialized literature review, it has not been addressed by the GGA. Moreover, the literature reveals that the methods applied to $R||C_{max}$ that have shown the best results have been exact algorithms and local searches. However, it is important to note that most of the proposals in the literature are focused on the design of this type of strategies, leaving aside the study of other metaheuristic algorithms. Finally, the state-of-the-art indicates that, although some genetic algorithms have been proposed for $R||C_{max}$, none of them have obtained good results by themselves and have been combined with local searches to improve their performance. This behavior is caused by the use of an inappropriate representation scheme to encode and manipulate the solutions. Therefore, we want to demonstrate that it is possible to design an efficient GGA for $R||C_{max}$ by using only traditional operators (crossover and mutation), without incorporating local search strategies, capable of competing with the state-of-the-art algorithms. In this sense, this work comprises the study and design of intelligent strategies of purpose-specific for the GGA operators that incorporate knowledge of the $R||C_{max}$ problem domain, employing exploratory data analysis techniques to identify and study the problem characteristics that influence its difficulty and the optimization process followed by the solution methods.

1.3 Justification

The design of high-performance metaheuristic algorithms responds to the need of organizations to be increasingly competitive and efficient since an important number of vital tasks for their proper functioning involve optimization problems that are difficult to solve. Many real-world optimization problems belong to the special class NP-hard, which implies that no efficient algorithms are known to solve them exactly in the worst-case [6]. Therefore, researchers throughout the world continue working on the design of high-performance heuristic algorithms. Since the solution to these problems implies a great challenge, the design of algorithms adapted to specific conditions and properties is usually the best option. This task has been facilitated through the implementation of characterization approaches that promote the understanding of the properties of the study problems and the optimization process presented by their solution methods.

In this research work, we apply state-of-the-art knowledge about the relationships between the optimization problem characteristics, the optimization process of the algorithms that solve them, and their final performance in order to generate knowledge of the problem Parallel-Machine Scheduling with Unrelated Machines and Makespan Minimization $R||C_{max}$. Finally, this work shows the importance of using a characterization approach, by employing the information gained from this study as a guide in the design of a high-performance GGA with a population initialization strategy, crossover and mutation operators, as well as the selection and the replacement mechanisms that use intelligent strategies that incorporate knowledge of the $R||C_{max}$ problem domain.

1.4 Thesis goals

1.4.1 Thesis main goal

Build robust, highly effective heuristic strategies that incorporate knowledge of the problem domain to solve the NP-hard grouping problem Unrelated Parallel-Machine Scheduling with Makespan Minimization $R||C_{max}$.

1.4.2 Thesis specific goals

- Adapt the GGA-CGT genetic operators (proposed to solve the Bin Packing Problem (BPP)) to solve the NP-hard grouping problem $R||C_{max}$ and analyze its potential in this problem.
- Conduct an experimental analysis for the characterization of the grouping problem $R||C_{max}$ and for the study of the algorithmic behavior presented by the heuristic strategies when solving $R||C_{max}$.
- Identify and implement new heuristic strategies that incorporate knowledge of the problem domain to solve the grouping problem $R||C_{max}$ efficiently.

- Conduct an experimental study of the optimization process of the implemented heuristic strategies applied to $R||C_{max}$.
- Develop new intelligent metaheuristic algorithms for $R||C_{max}$ that incorporate knowledge of the problem domain.

1.5 Thesis hypotheses

H_1. It is possible to identify the characteristics that impact the difficulty of the NP-hard grouping problem $R||C_{max}$ and understand how they affect the algorithmic behavior and the final performance of the metaheuristic algorithms that solve them.

H_2 It is possible to improve the performance of metaheuristic algorithms that solve the NP-hard grouping problem $R||C_{max}$ through the incorporation of efficient heuristic strategies designed using knowledge of the algorithmic optimization process.

1.6 Scope and limitations

1. The characterization approach to identify the properties that impact the difficulty of the instances will be applied only to the Parallel-Machine Scheduling Problem with Unrelated Machines and Makespan Minimization $R||C_{max}$.
2. The analysis of the optimization process will be conducted by taking into consideration the solution strategies designed in this research work and the best algorithms for $R||C_{max}$ relative to an exact method from the state-of-the-art.
3. Given the $R||C_{max}$ difficulty, for many instances of the specialized literature, the optimal solution is not known. Therefore, the comparative analysis of the performance presented by the state-of-the-art algorithms and the solution methods designed in this work will be conducted based on the results obtained by two hours of CPLEX (the high-performance solver of IBM ILOG, based on branch and cut methods) for each instance.
4. The identified factors could be a subset of the critical factors in the difficulty of an instance, the algorithm behavior of the solution methods, and their performance. Since it has been shown that even the most detailed and in-depth analyzes cannot guarantee the full identification of all possible factors.

1.7 Thesis organization

The document is organized as follows. Chapter 2 presents a brief review of the combinatorial optimization grouping problems and the state-of-the-art solution methods. Likewise, Chapter 3 introduces the problem under study, the NP-hard combinatorial optimization grouping problem parallel-machine scheduling with unrelated machines and makespan minimization, referred to as $R||C_{max}$ and a summary

of the results obtained by the best state-of-the-art algorithms. Later, Chapter 4 presents the first Grouping Genetic Algorithm (GGA) for $R||C_{max}$, highlighting its main features and strengths over other heuristic algorithms. On the other hand, Chapters 5, 6, 7, and 8 include a set of experimental studies to analyze and improve the optimization process of the GGA components: population initialization strategy, crossover operator, mutation operator, and reproduction technique, respectively. Similarly, Chapter 9 includes the approach, based on exploratory data analysis techniques, used to characterize the properties of $R||C_{max}$ instances and the optimization process of the GGA proposed based on the knowledge gained from the study of the GGA components in isolation in Chapters 5, 6, 7, and 8. Subsequently, Chapter 10 shows the main computational experiments used to analyze different aspects of the performance presented by the designed GGA for $R||C_{max}$. Thus, this section shows the way the GGA performance evolved, as well as a set of tests to assess the efficiency and robustness of the final version of the proposed GGA. Finally, Chapter 11 presents the conclusions obtained from this research work and the proposed paths of work.

Chapter 2

Grouping problems

Combinatorial optimization is a challenging research area with many real-world applications. Such applicability has motivated the scientific community to devote great efforts to generate useful knowledge to solve them. It is well-known that many combinatorial optimization problems (COPs) have high complexity, in such a way that in some cases, to date, no algorithm efficiently solves all their possible scenarios. Problems with the before-mentioned trait belong to the NP-hard class [6]. Over the last decades, numerous COPs with different characteristics have been identified. Therefore, the state-of-the-art includes several classifications that organize them according to their particular properties. One of the most studied COP classes is the well-known grouping problems class that, in general, implies to search an efficient distribution of a collection of items among a set of groups [7].

The grouping problems class includes combinatorial problems of challenging complexity, in such a way that most of them require a computationally expensive solution process. The solution of a grouping problem consist of finding an efficient partition of a set V with n items into a collection of D mutually disjoint subsets (groups) G_i, so that: $V = \cup_{i=1}^{D} G_i$ and $G_i \cap G_j = \emptyset$, $i \neq j$. In this way, Figure 2.1 shows a set V with $n = 10$ items divided into a collection of $D = 4$ groups. Solution methods for grouping problems seek for an efficient distribution of the n items into D ($1 \leq D \leq n$) distinct groups, such that the so-called objective function is minimized or maximized. This objective function is frequently formulated based on the structure of the groups, as well as the characteristics of the entire collection of groups. Furthermore, it is important to remark that all possible groupings are not allowed, since a set of constraints and conditions must be satisfied. In this order of ideas, some grouping problems are highly constrained, increasing their difficulty [8]. To facilitate the exploration of the search space of some grouping problems, metaheuristic algorithms can use cost functions that do not necessarily correspond to the objective functions of the problems. This action is helpful to address grouping problems where solutions with different partitions of the items can produce the same value of the objective function. Thus, the cost function allows differentiating the quality of two or more solutions that are different, but have the same value of the objective function. The cost function plays an important role in the GGA performance, since, generally, the search directions are generated based

on the characteristics of the best solutions. Therefore, it is essential to differentiate their quality. This approach mainly has been used to address the Bin Packing problem. However, the literature also includes other grouping problems handled in this way, like Graph Coloring and Task Assignment.

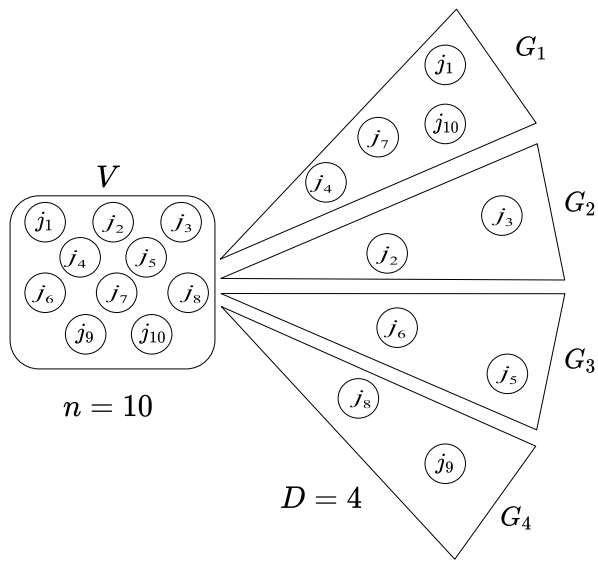


Figure 2.1: An example of a grouping problem with ten items distributed among four groups.

According to the scope of this literature review until 2019, the GGA has been used to solve forty NP-hard grouping problems, listed in Table 2.1. This chart contains the name of each grouping problem in the first column and some related works in the second one. The details of the general properties of most of these problems can be found in [7].

Table 2.1: Grouping problems addressed with GGAs. *Problem*: name of the problem. *Related works*: papers that introduce GGAs to solve each grouping problem.

Problem	Related works
Bin Packing	[8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
Blockmodel	[18, 19]
Carbon-Aware Distributed Cloud	[20]
Care Task Assignment	[21]
Cell Formation	[22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
Clique Partitioning	[32]
Clustering	[33, 34, 35, 36, 37, 38, 39]
Cutting Stock	[40]
Economy of Scale	[8, 41]
Equal Piles	[8, 42]
Estimating Discretionary Accruals	[43]
Facility Location	[44]

Continues in next page.

Problem	Related works
Vehicle Routing	[45, 46]
Feature Selection	[47, 48]
Job Shop Scheduling	[49, 50]
Graph Coloring	[51, 52]
Grouping Partners in Cooperative Learning	[45]
Handicapped Person Transportation	[53]
Home Healthcare Scheduling	[54, 55, 56]
Line Balancing	[45, 57]
Material Cutting	[58]
Maximally Diverse	[59]
Microcell Sectorization	[60]
Modular Product Design	[45, 61, 62]
Multiple Knapsack	[63, 64]
Multiple Travelling Salesperson	[65, 66, 67, 68]
Multiprocessor Scheduling	[69]
Multivariate Micro-aggregation	[70]
Order Batching	[45, 71]
Parallel-Machine Scheduling	[7]
Pickup and Delivery	[72]
Registration Area Planning	[73, 74, 75]
Reviewer Group Construction	[76]
Stock Portfolio	[77, 78, 79, 80, 81, 82, 83]
Supplier Selection	[45]
Task Assignment	[84, 85, 86, 87]
Team Formation	[45, 88]
Timetabling	[45, 51, 89]
Transition Path Generation	[90, 91]
WiFi Network Deployment	[92]

It emerges from Table 9.1 that some grouping problems such as Bin packing, Cell Formation, Stock Portfolio, and Clustering have many related works. This phenomenon can occur for two reasons. The first case is that some problems have several variants with different constraints and conditions, and there are GGAs to address some of them, such as Cell Formation [22, 25, 28, 29]. The second reason is that a problem has been addressed by several GGAs with different representation schemes, variation operators, and reproduction techniques. For example, the GGAs introduced in [9, 11, 15] to solve the Bin Packing problem.

Given the wide variety of grouping problems, A.H. Kashan *et al.* introduced a classification, based on the number of groups, the type of groups, and the dependence on the groups' order [4]. To create this classification, they took into consideration that the objective functions in the majority of the grouping problems are formulated based on the composition of the groups and the entire array of groups. In this way, the criterion number of groups allows arranging the problems as *constants* or *variables*. If the number of groups D is known, the problem is *constant*. Thus, the problem-solution

consists of finding an efficient assignment of the n items among the D given groups. Conversely, if the value of D is unknown, the problem is *variable*. Hence, the solution problem comprises the exploration of suitable groupings to minimize or maximize the number D of groups needed to allocate all the items n. On the other hand, the criterion type of groups allows dividing the grouping problem into *identical* and $non-identical$ problems. If the quality of a solution is modified, by exchanging all the items of two groups, that problem belongs to the $non-identical$ grouping class. Otherwise, the problem is part of the *identical* category. Finally, the criterion dependence on the groups' order arranges the grouping problems into *order dependent* and *not order dependent*. When the solution quality depends on the groups' order, the problem is *order dependent*. Consequently, grouping problems without such dependency belong to the *not order dependent* class [4].

During the literature review, we observed that some grouping problems included in Table 9.1 share a characteristic that, according to the scope of this study, has not been considered to classify them. That is the dependency on the order of the items in each group, like Cell Formation [22, 23, 24, 25, 26, 27, 28, 29, 30, 31] and Multiple Travelling Salesperson [65, 66, 67, 68]. In this class of problems, that we named *items order dependent*, the quality of the solutions can be modified when altering the position of the items within a single group.

It is important to note that some real applications can include problems related to the grouping of elements that do not meet the definition proposed by Falkenauer [93]. Such is the case of the lifetime maximization problems in Wireless Sensor Networks (WSNs), where the elements can belong to more than one group, allowing non-disjoint groups [94]. Currently, state-of-the-art GGAs cannot address these types of problems because they use operators designed to work with disjoint groups. In this way, if an item is in more than one group, it is removed from one of them to avoid non-disjoint groups. For future works, it could be interesting to study the performance of grouping operators adapted to solve this kind of problems.

Grouping problems occur in many practical applications, and there is a need for algorithms that solve them efficiently. The literature includes a great variety of solution methods, classified into two main groups, known as exact or approximate approaches. The exact algorithms guarantee to find an optimal solution in finite time by systematically exploring the search space. However, due to the complexity of NP-hard problems, the time required to solve them can grow exponentially in the worst case. Under these conditions, it is necessary to make use of approximate algorithms, also known as metaheuristic algorithms, which, although they do not guarantee to find the optimal solution, can obtain high-quality solutions in a considerably short computation time.

2.1 Metaheuristic algorithms in grouping problems

The specialized literature includes several algorithms designed to solve grouping problems using different approaches, like traditional mathematical methods, dynamic programming, simple heuristics, enumerative methods, and metaheuristics. In this

research, only the most representative metaheuristics are surveyed, comprising four neighborhood searches, seven evolutionary algorithms, as well as six swarm intelligence algorithms. In this way, this section presents the information thrown by google scholar using as search terms each before-mentioned "*metaheuristic*" for each "*grouping problem*" listed in Table 2.1 up to 2019. Metaheuristic methods have been broadly used to solve grouping problems, because of their general nature which allows them to be efficient in different problems without significant changes [95]. It is important to note that the state-of-the-art also includes hybrid metaheuristics (i.e., solution methods made up of a metaheuristic that works as the principal search motor and one local search that works in a second level) [96, 97, 98, 99, 100]. In this review, metaheuristics in hybrid or memetic algorithms are identified and considered individually. The readers interested in the development of hybrid metaheuristics are referred to [101] for useful guidelines.

2.1.1 Neighborhood searches

Neighborhood searches also known as trajectory searches work with a single solution. In general, the search process of metaheuristics designed using this approach consists of generating and exploring the neighbors of the current solution. Neighbor solutions are created using different techniques; for example, modifying the value of one variable or exchanging two or more elements. Hence, the neighborhood size depends on the strategy used to generate the neighbors. The performance of neighborhood searches has been examined by solving different grouping problems. For some problems, it has been demonstrated that simple local search methods can outperform sophisticated hybrid methods, producing high-quality solutions in a short time [102, 103]. An example is the work of Santos *et al.*, where the performance of different neighborhood searches designed for solving the unrelated Parallel-Machine Scheduling (PMS) problem with sequence-dependent setup times is investigated, analyzing different neighborhood structures, diversification and intensification strategies and parameter-tuning challenges [104]. For this review, four neighborhood metaheuristics are considered: Hill Climbing (HC) [105], Variable Neighborhood Search (VNS) [106], Simulated Annealing (SA) [107] and Tabu Search (TS) [108].

The Hill Climbing (HC) algorithm receives its name because it manages an iterative improvement strategy. In this form, the neighborhood of the current solution is used to increase (improve) its quality each cycle of the search process. Regularly, HC starts from an arbitrary solution (current solution). Then, iteratively tests new candidate solutions in the neighborhood of the current solution and adopts the new ones if they are better. HC techniques have been very efficient when they are combined with other approaches like dynamic programming and genetic algorithms to create hybrid and memetic algorithms. Until now, HC has been used to address thirteen of the twenty-two grouping problems contemplated in this review [105, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 100]. For example, Kato *et al.* introduced one of the last related works on solving grouping problems using HC [120]. In this work, the authors present a hybridization of HC with Particle Swarm Optimization (PSO) to address an extension to the traditional Job Shop Scheduling

(JSS) widely reported in the literature. The performance of the proposed algorithm was compared against other hybridizations of PSO and different local searches of the state-of-the-art algorithms, showing better results.

Similarly, Variable Neighborhood Search (VNS) is an iterative improvement search; however, this method explores increasingly distant neighborhoods, one at the time. Starting from an initial incumbent solution, a random solution is generated from the current neighborhood and a local search is applied to get a local optimum. If a new incumbent solution is found (the local optimum better than the current incumbent solution), the process is repeated starting with the first neighborhood. Otherwise, the process is repeated with the next neighborhood (which is typically larger). VNS has proven its efficiency in several grouping problems, working within cooperative approaches with exact techniques and as a part of hybrid algorithms combined with other metaheuristics. Until now, VNS has been adapted to tackle seventeen of the twenty-two grouping problems contemplated in this review [121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 106, 136]. Among the most recent efforts to address grouping problems using VNS is the work presented by Santos *et al.*, in 2019 [121]. In this study, the authors solve large instances of a new variant of the Bin Packing (BP) problem with a simple VNS, improving the results of sophisticated procedures.

On the other hand, Simulated Annealing (SA) takes as inspiration the chemical process of metal annealing. SA attempts to reduce the probability of becoming stuck in a local optimum by sometimes accepting neighbors with a lower quality than the current solution. In this algorithm, if a new solution in the neighborhood is better than the current solution, it will always replace it. But if the neighbors of the current solution are worse than the current solution, they are evaluated to decide probabilistically if a transition to a new solution is made or not. SA has been successfully applied to numerous grouping problems, and several modifications of the accepting rules and hybridizations with other metaheuristics have improved the performance of this algorithm. According to the scope of this review, eighteen of the twenty-two grouping problems contemplated in this review have been addressed by employing SA [107, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 133, 148, 149, 150, 151, 152]. In 2019, Leite *et al.* presented one of the last proposals to solve grouping problems with SA [146]. In this work, the authors assessed the performances of a fast SA to solve the Timetabling Problem (TP). They performed an experimental study solving the 2nd International Timetabling Competition (ITC 2007) benchmark set with SA. Experimental results indicate that SA improves on one out of twelve instances, and ranks third among the five best algorithms.

The last neighborhood search contemplated for this review is Tabu Search (TS), which uses the concept of memory and implements it through simple structures. TS works as an ordinary descent method that only permits moving to neighbors that improve the quality of the current solution. However, a short-term memory, known as the tabu list, stores recently visited solutions (or their attributes) to expand the local search and escape from local optimums. Thus, the neighborhood of the current solution is restricted, considering the solutions that do not belong to the tabu list. Over the years, several developments and refinements of TS have been proposed, including different

forms of memories, as well as hybridizations with other techniques. Thanks to these characteristics, TS has been used to deal favorably with nineteen of the twenty-two grouping problems contemplated in this survey [153, 154, 155, 108, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 118, 96, 166, 167, 168]. One of the last applications of TS for solving grouping problems is the work presented by Peng *et al.*, who addressed the Job Shop Scheduling (JSS) problem by combining TS with a genetic algorithm. Experimental results pointed out that this hybrid procedure has high optimization performance and practical value in the field of JSS [169].

It is important to note that the state-of-the-art holds other local search metaheuristic algorithms applied to solve grouping problems, like Greedy Randomized Adaptive Search Procedure (GRASP) [170, 171] and Iterated Local Search (ILS) [172, 173]. However, this review considers only the most representative, that is, the four already mentioned. Table 2 indicates the neighborhood algorithms (rows) used to solve each grouping problem (columns), according to the main research results in the field. Cells contain related work references, while empty cells indicate that we did not find any reference in specialized literature about it. Therefore, Tabu Search is the most used neighborhood search since it has been used to solve nineteen grouping problems. In contrast, Hill Climbing has been applied to solve thirteen problems only.

Table 2.2: Grouping problems addressed using neighborhood searches.

Metaheuristic	BP	LB	ALD	VR	CF	MTS	FL	JSS	TF	MPD	MM	MK	CS	TP	CP	MD	SS	GC	PMS	OB	HHC	SP
HC	[105]	[109]	[110]	[111]			[112]	[113]				[114]	[115]	[116]	[117]			[118]		[119]		[100]
VNS	[121]	[122]	[123]	[124]	[125]	[126]	[127]	[128]				[129]	[130]	[131]	[132]	[133]		[134]	[135]	[106]	[136]	
SA	[107]	[137]	[138]	[139]	[140]	[141]	[142]	[143]	[144]				[145]	[146]	[147]	[133]		[148]	[149]	[150]	[151]	[152]
TS	[153]	[154]	[155]	[108]	[156]	[157]	[158]	[159]	[160]			[161]	[162]	[163]	[164]	[165]		[118]	[96]	[166]	[167]	[168]

2.1.2 Evolutionary algorithms

Evolutionary algorithms are stochastic search methods inspired by the natural biological evolutionary process. These methods simulate some mechanisms of organic evolution, such as mutation, crossover, and natural selection. Until now, different evolutionary algorithms have been proposed, which use several variants of the before-mentioned mechanisms. Evolutionary algorithms used to address grouping problems, so far, include Genetic Algorithms (GA), Evolution Strategies (ES), Evolutionary Programming (EP), Genetic Programming (GP), and Differential Evolution (DE). Over the years, EAs have been widely applied to solve grouping problems with a good measure of success. An excellent overview of algorithms performance and current trends in EAs for Clustering Problems (CP) is presented by Hruschka *et al.* [174]. This paper discusses key issues on the design of EAs for data partitioning problems, such as usually adopted representations, evolutionary operators, and fitness functions.

The Genetic Algorithm (GA) is the most widely known evolutionary algorithm. The basic GA is very generic, and many aspects can be implemented differently according

to the problem. The process starts generating a random population of solutions. Then, for a certain number of generations, selected individuals are recombined and mutated to produce better solutions. First, a selection strategy is used to choose individuals, considering their fitness value. Next, the crossover operator is applied to the selected individuals to produce offspring, and these offspring are introduced to the population, employing a replacement strategy. Finally, some individuals are selected to be the subject of slight random perturbations through the mutation operator. The algorithm iterates a predefined number of generations, or until some stopping criterion, related to the problem or the algorithm performance, is met. Many variants of GAs have been developed and have been applied to solve a wide range of grouping problems, including GAs with different schemes for representation of solutions, selection, crossover, replacement, mutation, etc. This metaheuristic has shown promising results in solving grouping problems because it can incorporate new general or specific ideas easily. GAs have been hybridized with many other techniques, achieving high-quality results. Consequently, it has been used to solve the twenty-two grouping problems considered in this review [175, 176, 177, 178, 98, 179, 180, 181, 182, 183, 184, 185, 186, 116, 187, 133, 188, 189, 190, 191, 192, 193]. In [175] is presented one of the last applications of GA solving grouping problems. In this work, Laabadi *et al.* used GA to solve the a variant of Bin Packing (BP) problem, finding promising results. In [179], Zhu and Wu present another recent application of GA to solve grouping problems. In that work, the authors proposed an improved model for the optimization of Multiple Traveling Salesperson (MTS) problems with complex topology structure, the model was solved with the GA, showing excellent results.

Evolution Strategies (ES) paradigm was proposed originally by Rechenberg and Schwefel to work in continuous spaces, and later it was adapted to the discrete domain. In general, this evolutionary algorithm begins with a random population consisting of μ solutions; then, in each generation, the μ parents produce λ descendants through recombination and mutation, and the selection operator determines the μ fitter individuals to become the parents of the next generation. Initially, ES was designed to work with one parent and one child. And later, $(\mu + \lambda)$-ES and (μ, λ)-ES extensions were proposed to consider different selection schemes, giving rise to the self-adaptation of parameter mechanisms that encode strategy parameters directly onto the chromosome. In this way, (μ, λ)-ES uses a selection mechanism that only considers the λ newly generated offspring for the next generation, discarding the parents from the current generation even when they are better than all offspring. In contrast, $(\mu + \lambda)$-ES does consider both the offspring and the parents during the selection process. Nowadays, some efforts have been carried out using ES to solve eleven of the twenty-two grouping problems contemplated in this survey [194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204]. In 2018, Wang *et al.* presented one of the last studies related to the application of ES to solve grouping problems [199]. In this work, the authors analyzed the performance of ES in solving 6,000 random classic instances of a variant of the Team Formation (TF) problem, where ES reached the state-of-the-art results. Besides, Wang *et al.* studied the performance of ES addressing 1,556 realistic instances, where ES showed outstanding results.

Evolutionary Programming (EP) is quite similar to ES. Nevertheless, in EP, there is

no recombination operator because each individual corresponds to a distinct species; furthermore, the selection mechanism is different. In general, EP begins with a random population consisting of μ solutions; then, for a certain number of generations, the μ parent solutions of the current population are mutated to generate μ children. Next, parents and children compete in stochastic round-robin tournaments to be parents of the next generation. According to the state-of-the-art, EP has been used to address six of the twenty-two grouping problems contemplated in this review: Multiple Traveling Salesperson (MTS) [205], Cutting Stock (CS) [206], Cell Formation (CF) [207], Job Shop Scheduling (JSS) [208], Parallel Machine Scheduling (PMS) [209], and Clustering Problem (CP) [210]. For example, Chiong *et al.* used EP to solve the CS problem in [206]. In this work, the authors conducted an experimental study on solving benchmark problems to assess the performances of EP, getting promising results.

The paradigm of Genetic Programming (GP) adopts a similar search strategy as a GA for creating computer programs. In GP, the chromosomes of the population are not solutions as used in GAs, they are algorithms, represented by tree structures, that, when executed, allow to obtain candidate solutions to the problem at hand. GP starts with an initial population of randomly generated tree structures, iteratively evolved using special genetic operations adapted to work with tree structures. First, the fitness of each chromosome is evaluated, in terms of its performance, on the problem which it represents. Next, the variation operators are selected probabilistically for producing offspring, and a replacement strategy is applied to select the chromosomes for the new population. GP has allowed creating new techniques of solution, going from single assignment rules to more sophisticated methods like hyper-heuristics and hybrid metaheuristics. To date, GP has been used to address twelve of the twenty-two grouping problems contemplated in this review [211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222]. One of the grouping problems most studied using GP is the Job Shop Scheduling (JSS) problem [215]. Derived of this, Nguyen *et al.* summarise existing studies in this field until 2019 to provide new paths of work. They pointed out that the use of GP to solve JSS has contributed enough knowledge to the area. Nevertheless, they observed that there is still a lack of efficient representations of dispatching rules to enhance the effectiveness of GP to solve JSS.

Besides, Kenneth Price and Rainer Storn introduced the Differential Evolution (DE) algorithm in 1995. Originally, DE was designed to solve continuous problems. However, the state-of-the-art includes some adaptations of DE to deal with grouping problems. Its main characteristic lies in the variation operators since it uses vector differences to generate new solutions. DE begins generating an initial uniformly distributed random population of individual vectors. At each generation of the evolution process, for each vector in the population, also called target vector, mutation and crossover are applied to produce a trial vector. First, a mutant vector of the target vector is generated, employing a slight random perturbation. Next, the mutant vector and its corresponding target vector are recombined to generate a trial vector. Then, the selection operator compares the fitness of each trial vector to that of its corresponding target vector to determine which one will be maintained into the next generation. Due to its success in solving continuous problems, DE has been extended to solve grouping problems and combined with other metaheuristics in hybrid

methods obtaining effective and competitive results. Until now, DE has been used to solve thirteen of the twenty-two grouping problems contemplated in this review [223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235]. In [225], one of the most recent applications of DE to solve grouping problems is presented, addressing a real-world application of the Vehicle Routing (VR) problem. The central idea of this work is to design optimal routes minimizing the emission of direct greenhouse gases (i.e., carbon dioxide (CO2), methane (CH4), and nitrous oxide (N2O). From this study, the authors found interesting results for the design of routes with these characteristics.

In 1992, Emanuel Falkenauer introduced one of the most popular algorithms to solve grouping problems, the Grouping Genetic Algorithm (GGA) [93]. In general, this metaheuristic is an extension of the Genetic Algorithm that incorporates a group-based representation scheme. Thus, grouping problems are tackled considering the groups as the unit instead of the elements in each group. Similar to the Genetic Programming algorithm, variation operators must be adapted to work efficiently with group-based solutions encoding. GGAs have shown notable performance solving grouping problems; consequently, it has been employed to address twenty-one of the twenty-two problems surveyed [16, 236, 237, 238, 239, 65, 240, 49, 88, 241, 70, 64, 40, 242, 89, 59, 243, 37, 244, 56, 83]. In 2019, Singh and Sundar presented one of the most recent related works on solving grouping problems with Grouping Genetic Algorithms (GGA) [59]. In this work, the authors compared the performances of a GGA hybridized with a local search based on a swap strategy against a Tabu Search (TS) and a Variable Neighborhood Search (VNS) on solving the Maximally Diverse (MD) problem. Experimental results suggest that GGA reaches better results than TS and VNS, particularly for larger instances.

Finally, the success of GGAs in solving grouping problems motivated the development of the Grouping Evolution Strategies (GES). An extension of Evolution Strategies that incorporates the grouping representation scheme and a grouping mutation operator. GES is one of the most recent evolutionary algorithms. However, it has already been used to tackle Assembly Line Design (ALD), Bin Packing (BP), Order Batching (OB), Clustering Problem (CP), and the Parallel-Machine Scheduling (PMS) problem showing outstanding results [4, 245, 246, 247]. In 2018, Nejad *et at.* introduced one of the last applications of GES to solve a grouping problem [245]. In this work, the authors studied the performance of GES solving a variant of the ALD problem. They conducted an experimental study using test instances existing in the literature. Experimental results indicate that GES reaches the global solution of most problems of the high dimensional problems.

In the state-of-the-art, it is notable that there are other evolutionary algorithms used to solve grouping problems, such as Water Wave Optimization [248, 249]. However, this review considers only the most representative, that is, the seven already mentioned. Table 3 includes the evolutionary algorithms (rows) used to address each grouping problem (columns), according to the main research results in the field. Cells contain related work references. Thus, empty cells indicate that we did not find any reference in specialized literature about it. As it can be seen, Genetic Algorithms and Grouping Genetic Algorithms are the most used evolutionary algorithms since they have been used to solve twenty-two and twenty-one of the grouping problems considered in this

survey, respectively. In contrast, Evolutionary Programming and Grouping Evolution Strategies have been used to tackle only six and five grouping problems, respectively.

Table 2.3: Grouping problems addressed using evolutionary algorithms.

Metaheuristic	BP	LB	ALD	VR	CF	MTS	FL	JSS	TF	MPD	MM	MK	CS	TP	CP	MD	SS	GC	PMS	OB	HHC	SP
GA	[175]	[176]	[177]	[178]	[98]	[179]	[180]	[181]	[182]	[183]	[184]	[185]	[186]	[116]	[187]	[133]	[188]	[189]	[190]	[191]	[192]	[193]
ES			[194]	[195]	[196]		[197]	[198]	[199]					[200]	[201]	[202]			[203]			[204]
EP					[207]	[205]		[208]					[206]		[210]				[209]			
GP	[211]		[212]	[213]	[214]			[215]				[216]		[217]	[218]		[219]	[220]	[221]			[222]
DE	[223]		[224]	[225]		[226]	[227]	[228]				[229]		[230]	[231]		[232]	[233]	[234]			[235]
GGA	[16]	[236]	[237]	[238]	[239]	[65]	[240]	[49]	[88]	[241]	[70]	[64]	[40]	[242]	[89]	[59]	[243]	[37]		[244]	[56]	[83]
GES	[4]		[245]												[246]				[247]	[4]		

2.1.3 Swarm intelligence algorithms

Swarm intelligence algorithms emulate the collective self-organized behavior of natural systems for solving problems. In a swarm intelligence, a population of simple agents work together to produce computational intelligence. In recent years, these algorithms have been applied to a wide variety of grouping problems, showing excellent results. The performance of swarm intelligence strategies has been tested in solving different problems [250, 251, 252]. An example is the work of Milan *et al.*, where the advantages and disadvantages of nature-inspired metaheuristics for solving the Load Balancing (LB) problem are analyzed, identifying the most effective techniques [253]. To date, different algorithms have been proposed following this approach, including Ant Colony Optimization (ACO) [254], Particle Swarm Optimization (PSO) [145], Cuckoo Search (CS) [255], Artificial Bee Colony (ABC) [256] and Firefly Algorithm (FA) [257].

The ability of real ants to find the shortest path between their nest and a source of food was the inspiration to develop Ant Colony Optimization (ACO). This metaheuristic was designed to solve discrete problems, specifically for the traveling salesman problem. ACO encodes a given combinatorial optimization problem instance as a fully connected graph whose nodes are components of solutions, and edges are connections between components. A variable called pheromone is associated with each component, which can be read and modified by each ant. This value serves as a form of memory. At each iteration of the algorithm, several artificial ants are considered. Each of them builds a solution to the problem, step by step, adding a feasible solution component, according to a stochastic mechanism that is biased by the pheromone and heuristic information about the problem. Next, the pheromone values are updated to make solution components belonging to good solutions more desirable for ants in future iterations. Many variants of ACO have been created and applied to a wide range of discrete problems, showing outstanding results. The aforementioned motivated the use of ACO in twenty of the twenty-two grouping problems contemplated in this review [258, 259, 260, 261, 95, 262, 263, 264, 265, 266, 267, 254, 268, 269, 188, 270, 271, 272, 273, 274]. One of the last proposals on solving grouping problems with ACO is presented by Selvakumar and Guanasekaran, who introduced an enhanced ACO to solve the Load Balancing (LB) problem [259]. Experimental results indicate that the proposed algorithm has a

significant improvement over traditional algorithms, with respect to average execution time, average response time, and total cost.

In contrast, Particle Swarm Optimization (PSO) is a metaheuristic created to optimize continuous search spaces. This metaheuristic was designed using as inspiration the swarming and flocking behaviors in animals. PSO begins generating a population of particles with random positions and velocities on the search space. Each particle is a candidate solution to the problem, defined by three vectors in the d-dimensional search space: a velocity vector, a position vector, and a memory vector, which helps it in remembering its fittest known position discovered so far. At each iteration, directed transformations move each particle in the search space in response to discoveries obtained from the environment. First, the velocity of the particle is dynamically adjusted, considering its fittest known position and the position of the best particle among all the particles in its topological neighborhood. Next, the position of the particle is updated, adding the velocity vector to the position vector. Finally, the fitness of the particle is evaluated and, if necessary, the best-discovered locations are updated. Although PSO is planned to deal with continuous domains, it has also been adapted to tackle twenty of the twenty-two grouping problems contemplated in this review [275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 145, 286, 287, 288, 289, 290, 272, 291, 292]. In [277] is presented one of the most recent applications of PSO to solve grouping problems. In this work, PSO was applied to solve a variant of Assembly Line Design (ALD) problem. Experimental results suggest that the proposed approach reached good solutions for all test instances within a short computational time.

In the same way, Xin-She Yang and Suash Deb introduced the Cuckoo Search (CS) metaheuristic in 2009. They took as inspiration the behavior of some cuckoo species during the breeding stage, that lay eggs in a host's nest, removing others' eggs to increase the hatching probability of their own eggs. In the optimization context, each egg in the nest represents a solution, and the cuckoo's egg represents a new solution. When generating a new solution (a cuckoo's egg), a Lévy flight is performed, which essentially provides a random walk. CS starts generating a population of host nests, then for a certain number of iterations, new and potentially better solutions replace solutions in the nests. In each iteration, a new solution is generated, and its quality is compared with another random old solution in the population. In the case the new solution is better, it will replace the old solution. Therefore, a fraction of the worst solutions is replaced by new solutions. According to the scope of the literature review, fifteen of the twenty-two grouping problems contemplated in this review have been addressed employing CS [293, 294, 295, 296, 297, 298, 299, 300, 301, 255, 302, 303, 304, 97, 305]. In 2019, Karoum and Elbenani analyzed the performances of CS in solving the Cell Formation (CF) problem [297]. The results indicate that this metaheuristic is suitable for addressing this problem since it can reach 32 out of 35 benchmark problems (91.43%).

Similarly, the collective behavior of honey bees during the search and exploration of food sources was used by Karaboga in 2005 to develop the Artificial Bee Colony (ABC). Thus, three types of solutions (employee, onlooker, and explorer) are used to carry out an efficient search process. According to the state-of-the-art, sixteen of the twenty-two grouping problems contemplated in this review have been tackled using ABC [306,

307, 308, 309, 256, 310, 264, 311, 312, 313, 314, 315, 316, 317, 318, 319]. One of the most recent works on solving grouping problems using Artificial Bee Colony (ABC) is presented by Davoodi *et al.*, in 2019. In this work, the authors hybridized ACO with a Genetic Algorithm (GA) to solve the Vehicle Routing (VR) problem [308]. Results suggest that the production of the explorer bees is the most relevant factor in the search process of the proposed metaheuristic for solving VR.

Another swarm intelligence used to solve grouping problems is the Firefly Algorithm (FA), which is inspired by the social communication of fireflies via luminescent flashes and their synchronization. In this metaheuristic, a solution is represented by a firefly with a brightness associated that corresponds to its quality, and each firefly attracts its partners proportionally to its brightness. FA starts creating a population of fireflies (solutions) randomly distributed in the search space. Then, for a certain number of iterations, fireflies will move toward other positions, finding potential candidate solutions. For each firefly in the population, its brightness is compared with all other solutions, and its position is updated considering brighter fireflies. To date, FA has been used to tackle fifteen of the twenty-two grouping problems contemplated in this review [320, 321, 322, 323, 324, 325, 326, 327, 328, 257, 329, 330, 252, 331, 332]. In 2018, Ezugwu and Akutsan presented one of the last related works on solving grouping problems applying FA [252]. In this work, the authors addressed a variant of the Parallel Machine Scheduling (PMS) problem with FA. Experimental results indicate that the performance of the proposed FA is competitive, fast, and efficient for both small and large problem instances.

The last swarm intelligence technique considered in this review is the Grouping Particle Swarm Optimization (GPSO), an extension of the traditional Particle Swarm Optimization that incorporates the grouping representation scheme, as well as a grouping variation operator. Until now, only five of the twenty-two grouping problems contemplated in this review have been addressed using this technique, because it is a relatively new algorithm [333, 56, 334, 335]. However, it has shown promising results. In 2016, Xu proposed one of the last applications of GPSO on solving grouping problems. In this work, the author studied the performances of GPSO to solve ten real instances from the industry and ten more randomly generated of a variant of the Cutting Stock (CS) problem, showing interesting results.

It is important to remark that the state-of-the-art includes other swarm intelligence algorithms used to solve grouping problems, like Biogeography-Based Optimization (BBO), applied to solve Bin Packing and Graph Coloring [336, 337]. However, this review considers only the most representative, that is, the six already mentioned. Table 4 shows which swarm intelligence algorithms (rows) have been used to tackle each grouping problem (columns). Cells contain related work references. As can be seen, Ant Colony Optimization and Particle Swarm Optimization are the most popular swarm intelligence algorithms since they have been used to tackle twenty grouping problems. In contrast, only five problems have been addressed employing the Grouping Particle Swarm Optimization since it is a relatively new metaheuristic.

Table 2.4: Grouping problems addressed using swarm intelligence metaheuristics.

Metaheuristic	BP	LB	ALD	VR	CF	MTS	FL	JSS	TF	MPD	MM	MK	CS	TP	CP	MD	SS	GC	PMS	OB	HHC	SP
ACO	[258]	[259]	[260]	[261]	[95]	[262]	[263]	[264]	[265]		[266]	[267]	[254]	[268]	[269]		[188]	[270]	[271]	[272]	[273]	[274]
PSO	[275]	[276]	[277]	[278]	[279]	[280]	[281]	[282]	[283]	[284]		[285]	[145]	[286]	[287]		[288]	[289]	[290]	[272]	[291]	[292]
CS	[293]	[294]	[295]	[296]	[297]		[298]	[299]		[300]			[301]	[255]	[302]		[303]	[304]	[97]			[305]
ABC		[306]	[307]	[308]	[309]	[256]	[310]	[264]	[311]			[312]			[313]	[314]	[315]	[316]	[317]	[318]		[319]
FA	[320]	[321]	[322]	[323]	[324]	[325]	[326]	[327]				[328]		[257]	[329]			[330]	[252]		[331]	[332]
GPSO	[333]							[335]					[334]							[333]	[56]	

2.2 Conclusions of the literature review

As a result of the literature review, seventeen metaheuristics and twenty-two grouping problems were reviewed, comprising four local search strategies, seven evolutionary algorithms, and six swarm intelligence algorithms. The study revealed that the metaheuristic used to solve the largest number of grouping problems is the genetic algorithm. Figures 2.2 and 2.3 show a graphical comparison of the literature review summarized in Tables 1-4. Figure 2.2 shows a graphical comparison of the number of grouping problems (indicated on the horizontal axis) addressed by each algorithm (depicted on the vertical axis).

From this plot, one can see that Tabu Search (TS) is the local search used to address the greatest number of the grouping problems considered in this review since it has been applied to solve nineteen of the twenty-two problems. In contrast, Hill Climbing (HC) only has been used to address thirteen problems. Regarding swarm intelligence algorithms, Figure 2.2 indicates that Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO) have been the metaheuristics used to solve the greatest number of problems, applied to solve twenty of the twenty-two problems considered. Finally, Figure 2.2 indicates that from the seventeen selected metaheuristics, the evolutionary algorithms: Genetic Algorithms (GA) and Grouping Genetic Algorithms (GGA) have been used to solve a wider range of the problems, applied to solve twenty-two and twenty-one problems, respectively.

On the other hand, the detail of the number of algorithms (horizontal axis) used to address each grouping problem (vertical axis) is presented in Figure 2.3. From this plot can be seen that some grouping problems have been investigated using a wide range of metaheuristic algorithms, including Job Shop Scheduling (JSS) and Clustering Problem (CP). In contrast, this picture also shows that there are scarce studies to solve problems, like Modular Product Design (MPD), and Multivariate Microaggregation (MM).

In addition to this, Figure 2.4 includes the number of results thrown by google scholar about the term “metaheuristic" for each grouping problem listed in Table 1 from 2015 to 2019. This graph suggests that there have been several studies on solving problems like Bin Packing (BP), Load Balancing (LB), Vehicle Routing (VR), Facility Location (FL), and Job Shop Scheduling (JSS) using metaheuristics over the last years, highlining the clear intensification on the study of the Vehicle Routing (VR) problem. In contrast, Figure 2.4 also shows the least studied grouping problems, including Multivariate

Microaggregation (MM), Modular Product Design (MPD), Maximally Diverse (MD), and Stock Portfolio (SP). Such problems can be seen as an opportunity niche since there are many metaheuristics that have not been used to solve them. So it would be interesting to know their performance addressing them.

The state-of-the-art suggests that problems with a higher occurrence in real-world applications are the most studied ones. For example, sixteen metaheuristics have been used to address the Clustering Problem (CP). This problem has received increasing attention since it occurs in practical problems as off-line and online search engines, voice and data mining, pattern recognition, image processing, bioinformatics, machine learning, and reports analysis [338]. Another grouping problem quite investigated is the Timetabling Problem (TP), which takes place in nurses schedules, sports schedules, transportation schedules, university schedules, among many other applications [338]. Finally, as Figure 2.4 indicates, another widely studied grouping problem is Vehicle Routing (VR), which plays a central role in the fields of physical distribution and logistics. Over the last years, this problem has been significantly studied. As a result, nowadays, there are a wide variety of variants of VR, metaheuristics that solve them, and extensive literature about it [124, 178, 278].

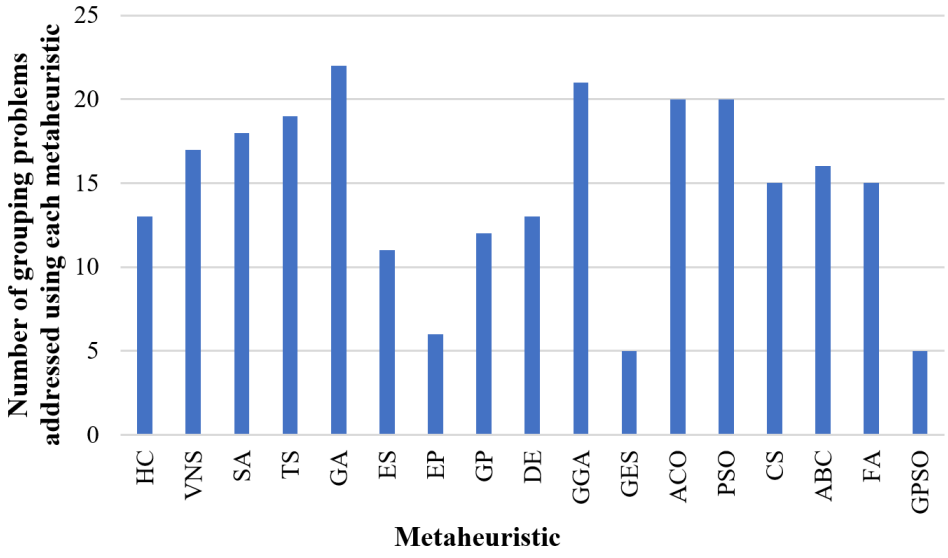


Figure 2.2: Comparative graph of the number of grouping problems addressed using each metaheuristic.

The literature review also allowed to identify possible directions for the development of more efficient metaheuristic techniques to solve NP-hard grouping problems. Some important issues are discussed below.

a) In recent years, some efforts have been carried out by proposing and adopting new search methods to solve grouping problems; for example, Evolution Strategies (ES) and Evolutionary Programming (EP) in evolutionary computation, as well as the Firefly Algorithm (FA) and Cuckoo Search (CS) regarding the swarm intelligence

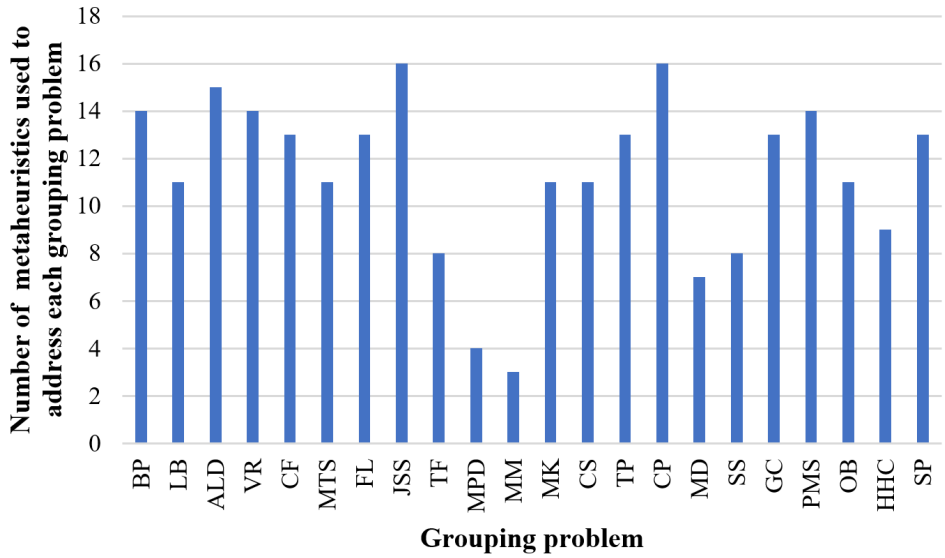


Figure 2.3: Comparative graph of the number of metaheuristics used to address each grouping problem.

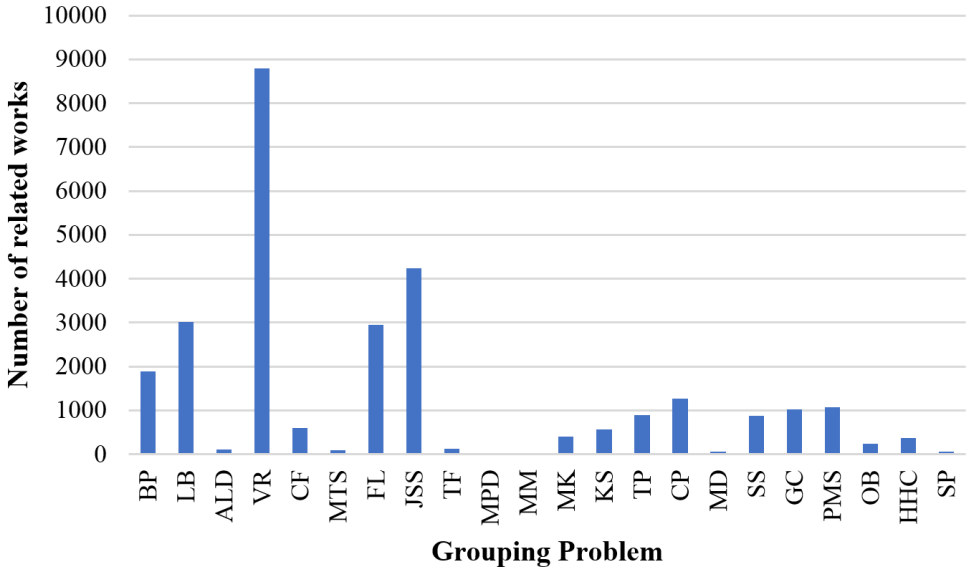


Figure 2.4: Results threw by google scholar about the terms “metaheuristic" for each “*grouping problem*" in Table 1.

approach. That is, there exists a trend to explore the algorithmic behavior of new metaheuristics addressing grouping problems. However, none of the metaheuristics in the state-of-the-art has been analyzed to explain the reasons for good or bad behavior in different grouping problems instances.

b) The current trend is to focus on solving real-world problems, mainly those directly affecting the society, such as the case of applications related to health. Nevertheless, for most of the real-world problems, it is unclear how to select and integrate the appropriate techniques to solve them and what is the expected performance of different strategies.

c) The group-based representation scheme can be used to improve the performances of other metaheuristics that solve grouping problems, besides Genetic Algorithms (GA). But only Particle Swarm Optimization (PSO) and Evolution Strategies (ES) have been adapted to deal with this issue. Not much work has been done on the performance study of these new grouping metaheuristics to different grouping problems.

d) One of the main challenges in the development of high-performance algorithms for grouping problems is the design of efficient strategies that work together with the grouping encoding scheme and the features of the grouping problems instances to find the optimal groups in fewer function evaluations.

The conclusions obtained from the analysis of the seventeen metaheuristic algorithms used to address the twenty-two NP-Hard grouping problems gave the guideline to establish the objectives of this research project. We observed that (1) Grouping Genetic Algorithms (GGA) is the metaheuristic used to solve the widest range of grouping problems; (2) the GGA had not been used to address the NP-hard grouping problem Parallel-machine scheduling with unrelated machines and makepan minimization ($R||C_{max}$); (3) in recent years some efforts have been made proposing and adopting new search methods to solve grouping problems. However, there are few studies on the reason for the good or bad behavior of these proposals.

Therefore, in Chapter 4 we present the first GGA for $R||C_{max}$, Chapters 5, 6, 7, and 8 include systematical studies to analyze in isolation the algorithmic behavior of each heuristic strategy used by GGA. That is the population initialization strategy, crossover, mutation, and reproduction technique. The knowledge gained from these studies will be used to design an Enhanced GGA (EGGA). In this way, Chapter 9 contains an experimental study to characterize the structure of the $R||C_{max}$ instances and the EGGA algorithmic behavior, looking for possible improvements in its performance. Thus, the knowledge of the problem domain obtained will be used to design the Final GGA (FGGA). Finally, Chapter 10 includes a set of tests to analyze the efficiency and robustness of the FGGA.

Chapter 3

The $R||C_{max}$ problem

This chapter introduces the problem under study, the NP-hard combinatorial optimization grouping problem parallel-machine scheduling with unrelated machines and makespan minimization, referred to as $R||C_{max}$. In this way, this section includes a general description of the problem, highlighting its main characteristics and the mathematical model considered in this research. Moreover, it contains a detailed description of the benchmark of test instances used to evaluate the performance of the algorithms developed to solve the $R||C_{max}$ problem. Finally, this section presents a survey of the solution methods for $R||C_{max}$ in the specialized literature, standing out the results obtained by the best state-of-the-art algorithms.

The statement of the classical Parallel-machine scheduling problem can be generalized as follows. Scheduling a collection of n jobs $N = \{j_1, ..., j_n\}$ in a set of m machines $M = \{i_1, ..., i_m\}$, in such a way that each machine i can process one job at the time, and every job is assigned to exclusively one machine. In this sense, the aim is to find the schedule that optimizes a certain performance measure [339].

Parallel-machine scheduling is, in fact, a family of problems. The definition of these problems involves the specification of several parameters, like the resource environment, job characteristics, optimization criteria, and scheduling environment, among others. The $\alpha|\beta|\gamma$ notation helps to differentiate the problem variants in the specialized literature [340, 341]. The first variable, α is used to indicate the machine environment (i.e., a problem with a single machine ($\circ$), identical parallel machines (P), uniform parallel machines (Q), unrelated parallel machines (R) or an open shop (O)). The second one, β specifies the job characteristics (i.e., no preemption ($\circ$), limited resources (res), a precedence relation (r_j), and the processing time (p_j or p_{ij})). Finally, γ defines the goal of interest (i.e., the completion time (C_i), the lateness (L_j), the tardiness (T_j), and the unit penalty (U_j)).

This work focuses on the most general variant of Parallel-machine scheduling problems, the $R||C_{max}$ problem, consisting of unrelated machines, jobs without preemption, and minimization of the maximum completion time, commonly referred to as makespan. $R||C_{max}$ can be considered an assignment problem since the ordering process of the jobs in the machines does not affect their performance. Given the above, $R||C_{max}$ is

stated as a formulation of Mixed Integer Linear Programming (MILP) as follows [342]:

$$\min C_{max} \tag{3.1}$$

$$\sum_{i=1}^{m} x_{ij} = 1 \ \forall j \in N \tag{3.2}$$

$$\sum_{j=1}^{n} p_{ij} \cdot x_{ij} \leq C_{max} \ \forall i \in M \tag{3.3}$$

$$x_{ij} \in \{0, 1\} \ \ \forall j \in N, \forall i \in M, \tag{3.4}$$

where $C_{max} = max(C_i)$, C_i indicates the completion time that each machine i needs to process its assigned jobs; p_{ij} is the processing time of job j on machine i; and $x_{ij} = 1$ if the job j is assigned to the machine i, otherwise $x_{ij} = 0$.

According to the specialized literature, $R||C_{max}$ holds a NP-hard complexity since it was shown that $P||C_{max}$, a more simple scheduling problem, belongs to that class [6]. In addition to this, in 1990 Lenstra *et al.* showed that it is not possible to develop an algorithm with a better worst-case ratio approximation than 3/2 to solve $R||C_{max}$ unless $P = NP$ [343].

Due to the challenge of solving $R||C_{max}$, the specialized literature includes a wide variety of solution methods designed under different approaches, covering exact methods, two-phase algorithms, local searches, evolutionary algorithms, and hybrid algorithms. The next section presents a summary of the state-of-the-art proposals.

3.1 Solutions methods for $R||C_{max}$

The history of this $R||C_{max}$ problem begins in 1974, when Bruno, Coffman, and Sethi propose the $2R||C_{max}$ problem at the University of Pennsylvania. That is, the problem addressed in this research project but considering exactly two machines [344]. Moreover, they present the first deterministic algorithm to solve this problem. Two years later, Horowitz and Sahni develop the second deterministic and the first approximate method to solve the same problem $2R||C_{max}$ [345].

It was until 1977 that Ibarra and Kim propose the first heuristics for $R||C_{max}$ that are still widely used up to now, referred to as heuristic A, B, C, and D [346]. The four heuristics use the assignment heuristic Min() and implement different strategies to sort the jobs, based on the randomness, the shortest time $\min(p_j)$ in which each job j can be processed, and the longest time $\max(p_j)$ in which each job j can be processed. Given a job j, Min() calculates the processing time C_i that each machine i would have if it is assigned the job j, using the equation $C_i = C_i + p_{ij}$. Finally, it identifies the machine i that can process its assigned jobs plus job j faster than the others. In this way, each heuristic uses a different strategy to sort the jobs, to later allocates them by applying the heuristic Min().

Three years later, De and Morton perform an analysis of the opportunities and weaknesses of the algorithms proposed by Ibarra and Kim in [346]. They used the knowledge gained to introduce three heuristics, referred to as E, F, and G [347]. The heuristic E uses the average(p_j) criterion to sort the jobs and use the assignment heuristic Min(). On the other hand, the heuristic F calculates the makespan of the problem to solve with the heuristics B, C, D, and E, introduced by Ibarra and Kim [346], and selects the heuristic that generates the shortest makespan. Finally, the heuristic G uses the lower bound βC_{max} which helps machines fill up in a balanced way together with the heuristics E and Min().

A similar approach is introduced by Davis and Jaffe in the 1980s, based on the efficiency of each machine i to process every job j [348]. In this work, they present several solution methods that work similarly to the heuristics introduced in [346, 347]. These algorithms have the peculiarity that they use the alternate list $Efficiency$ that saves the efficiency of each machine i to process each job j, used to establish different criteria to assign jobs. For example, prohibiting assigning a job to the machine that processes it slowest.

On the other hand, Lawler *et al.* present a survey of the heuristics developed up to 1982 for $R||C_{max}$ [341]. Furthermore, in this work, they introduce a problem relaxation, stating that a feasible solution can be generated by adding the constraint in Equation 3.5. To solve this relaxation of the problem, the variables where $p_{ij} > d$ are eliminated from the system of equations to subsequently solve the relaxation. If the solution is infeasible, the value of d is increased; whereas, if the solution is feasible, the value of d is reduced. The lower possible value of d is identified by means of a binary search, considering that the value of d found must generate a feasible solution.

$$\sum_{j=1}^{n} p_{ij} x_{ij} \leq d \qquad \forall i \in M \tag{3.5}$$

Another important date for the $R||C_{max}$ problem is 1985 because in this year Pots introduces the first two-phase algorithm, referred to as Linear Programming and Enumeration (LPE) [349]. In the first phase, LPE generates a partial solution by means of a relaxation of the original linear programming problem. Subsequently, LPE uses the second phase to transform the partial solution into a feasible solution (if necessary) by assigning the remaining jobs, known as fractional jobs, using enumeration methods like the heuristics A, B, and C, described above.

In 1990 Lenstra *et al.* propose to use the rounding theorem in two-phases [343]. The central idea of the authors is to divide jobs and machines based on a certain time threshold ϵ. In addition, a deadline is assigned to each machine. In this way, the machines cannot exceed a certain processing time. Additionally, the authors present a demonstration to reaffirm that no algorithm can solve $R||C_{max}$, in polynomial time.

After six years of the first two-phase heuristics, Hariri and Potts present five new two-phase heuristics (LP/H) in 1991 [350]. The work includes two types of LP/H, known as integrated and non-integrated. In a non-integrated two-phase heuristic, the second phase allocates the fractional jobs using linear programming without taking into account the first phase's scheduling; while in the integrated heuristics, the second phase

bases its decisions on the total processing time assigned to each machine in the first phase.

The next year, Van de Velde proposes a branch-and-cut algorithm and an iterative local search, based on a surrogate and dual relaxation of the problem [351]. The relaxation of the problem consists of replacing the constraint (1) of the original problem with a constraint that incorporates a vector of Lagrange multipliers [$\lambda_{i=1}$, ..., λ_m].

In 1994 Glass *et al.* conduct a study to design an efficient metaheuristic for $R||C_{max}$, comparing the performance of a standard GA, the neighborhood-based methods Descent Algorithm (DA), Tabu Search (TS), and Simulated Annealing (SA), as well as a memetic algorithm made up of GA and DA, called Genetic Descent (GD). Experimental results highlight the GD performance [352].

Likewise, Piersma and van Dijk study the Descending Iterative Local Search (DILS) and Tabu Search (TS) two years later [353]. From this work outstands the way the neighborhood is generated, based on the efficiency of the machines. The experimental results show that this criterion (machine's efficiency) helps to reach better solutions.

One year later, Martello *et al.* introduce a set of lower bounds based on the Langrangian relaxation to the $R||C_{max}$ problem, proposed by Lawler *et al.* in 1982 [341]. They use the generated limits to develop an approximate algorithm and a Branch and Bound algorithm, providing a good computational performance [354].

In 1998 Srivastava introduces another implementation of the Tabu search (TS) for $R||C_{max}$, called Modified Tabu Search Heuristic (MTSH) [355]. This algorithm uses hashing techniques that control the aspiration test (which seeks to remove jobs from the machines with more processing time C_i to re-assign them to the machines with less C_i, attempting to reduce the makespan) and tabu restrictions (to avoid movements that lead to solutions already visited). Experimental results show that MTSH can obtain quality solutions to problems of practical size.

2002 is one of the most productive years for $R||C_{max}$ problem. The literature includes three papers from this year [356, 357, 358]. First, Mokotoff and Chrétienne introduce a cut plane scheme to generate an approximate algorithm and a Branch and Bound [356]. The experimental results indicate that the branch and bound algorithm obtains the best results, showing an efficient performance in most of the case studies. On the other hand, Serna and Xhafa develop a parallel approximate algorithm that uses a relaxation to the positive linear problem. This algorithm generates fractional solutions (within the limits of feasibility). Therefore, it uses a random rounding strategy that transforms infeasible solutions into feasible ones [357]. Finally, Mokotoff and Jimeno present three branch-and-cut approaches using a MILP formulation. This work also establishes partial enumerations considering the integrality of a part of the set of binary variables by using the characteristics of each problem. The experimental studies showed that this type of approach has a performance that improved the results of the state-of-the-art up to that moment [358].

Two years later, Guo *et al.* perform another effort to analyze the performance of metaheuristics like SA, TS, and the Squeaky Wheel Optimization (SWO) algorithm. Furthermore, the authors present improvements to the local searches, TS and SA, and design a new lower bound, called $C2$, that uses the smallest and second-smallest

processing time for each job. Finally, the knowledge obtained from this study is used to design a hybrid algorithm formed by SWO and the improved TS, showing significant results [359].

Another important date for $R||C_{max}$ is 2004. In this year, Ghirardi and Potts introduce an improvement to the Recovering Beam Search (RBS) method, which allows it to return to previously visited solutions. RBS is a truncated Branch and Bound algorithm where only the best nodes are selected from each branch. RBS was well received by the scientific community because it obtains good results in large instances (over 50 machines and 1000 jobs). In addition, in the same year, Pfund *et al*. present a survey on the deterministic problem of PMS, concluding that $R||C_{max}$ has been relatively little studied compared to other research areas [360].

The following year, Kumar *et al*. propose a rounding algorithm using linear algebra and randomization, based on SchedRound that offers a unified way to address a number of different goals in job scheduling with unrelated parallel machines [361]. The main virtue of this approach is that it can be embedded in different search strategies to improve its performance.

In 2007, Aburas proposes a local search to solve a real-world application of $R||C_{max}$, showing a good performance by solving a problem involving the fabrication of roof trusses at a major home building company. In addition, in this year, Gairing *et al*. present a formulation of $R||C_{max}$ as a generalized flow problem in a bipartite network, solved with a generic algorithm of minimum cost flow, showing unpromising results [362].

Another interesting proposal is presented in 2008 [363]. This work presents a set of grouping techniques, where the basic idea is to reduce the number of jobs as follows. The data is first rounded in such a way that a constant number of different job profiles results. Subsequently, jobs with the same profile are merged (grouped) to form new jobs and reduce the initial number of jobs. Finally, dynamic programming or enumeration methods are used to build the solutions.

The next year, Lin *et al*. introduce another two-phase heuristic, called LP/Roundup [364]. The first phase of LP/Roundup is based on the relaxation of the problem proposed by Potts in [346]. The fractional jobs are assigned based on the efficiency of each machine to process every job. In addition to this, LP/Roundup uses a procedure to reduce the makespan of the generated solution by rearranging the jobs on the different machines.

Another productive year for the $R||C_{max}$ problem is 2010 since four papers are presented this year. First, Fanjul-Peyro and Ruiz propose a benchmark with the 1,400 instances, used in this work, described in detail in Section 3.2. Furthermore, they present the results obtained by two hours of ILOG CPLEX (version 11.0) for the 1400 instances, solving about 34% of the instances optimally. Finally, they develop a set of simple algorithms, based on iterative greedy search algorithms (IG) and two neighborhood models giving rise to the algorithm NVST-IG+, which showed a highly competitive performance, that exceeded even the best state-of-the-art algorithms in most of the 1400 instances studied [342].

On the other hand, Sivasankaran *et al.* present a two-stage heuristic. This heuristic uses the first stage to generate a complete solution by assigning each job j to the fastest machine i to process it. Thus, in the second phase, it tries to improve the solution by performing interchanges between the jobs of each machine [365]. Furthermore, they present another work with a comparison between the local search SA and randomized Adaptive Search Procedure (GRASP). The experimental results indicate that SA performs better than GRASP [366].

Finally, the last related work of 2010 presents a Variable Neighborhood Descent (VND). This algorithm avoids stalling at local optima by incorporating neighborhoods of different sizes and incorporating a second objective (measuring the processing efficiency of each job j on each machine i) used to guide the search. VND showed a competitive performance on solving the test instances used [367].

The next year, Fanjul-Peyro and Ruiz present another interesting work, that consists of a set of methods to reduce the size of instances [368]. The main idea of these methods is to reduce the number of machines that can process a job based on their efficiency. Thus, for example, given a problem of 100 jobs and 10 machines, for each job j, the k machines with the shortest time required to process j are selected. They incorporate these reduction methods into different algorithms and assessed them using the 1400 instances proposed by themselves [342], showing a highly competitive performance that outperformed the best state-of-the-art algorithms in most of the study cases.

On the other hand, in 2011 Lin *et al.* present one of the few proposals related to the application of Genetic Algorithms (GA) to solve the problem $R||C_{max}$. In this work, the authors improve the standard GA by incorporating the job-based representation scheme and symbols and variation operators that work with this type of encoding. This algorithm shows highly competitive results on randomly generated test instances [369].

Finally, in 2015, Sels *et al.* present the last work related to $R||C_{max}$. They introduce two metaheuristics, a GA and a TS, hybridized with a Branch and Bound procedure and a local search algorithm [96]. The algorithms are tested in the 1400 test instances of Fanjul-Peyro and Ruiz. The experimental results indicate that the hybridization of the GA with the branch and bound algorithm has highly competitive results, exceeding the best state-of-the-art algorithms in some studied cases.

Given the complexity of $R||C_{max}$, several solution methods have been proposed throughout history under different approaches. Figure 3.1 includes a plot that allows graphically observing the number of related works presented for each approach. The $x-$axis represents each approach, while the $y-$axis indicates the number of related works found in the specialized literature.

As can be seen in Figure 3.1, the largest number of works are related to two-phase methods [343, 349, 361, 364, 370, 371, 372, 357] and local searches [352, 96, 352, 353, 359, 352, 359, 373, 367, 353, 373, 351, 342, 365] with eight and fourteen papers, respectively. The state-of-the-art also includes four works related to deterministic heuristics [345, 346, 347, 348] and five for exact methods [96, 354, 356, 358, 374]. Finally, the approaches with fewer proposals are evolutionary algorithms [96, 352, 375] and hybrid algorithms [352, 96] with three and two related works, respectively.

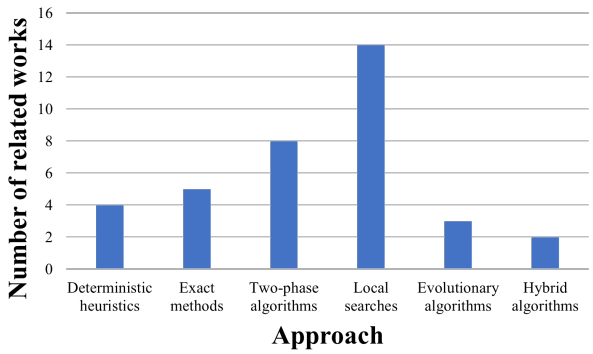


Figure 3.1: Related works for each approach used to design solution methods for $R||C_{max}$.

Table 3.1 shows the main characteristics of the best state-of-the-art algorithms for $R||C_{max}$, including the Partial enumeration, the Recovering Beam Search (RBS), the Iterative Greedy local search (NVST-IG+), and the Hybrid Tabu Search (HTS) [96]. The first column indicates the aspect analyzed from each work, covering the Approach, Author, Hybridization, $R||C_{max}$ heuristics, Search strategies, Strategies to modify a solution, Strategies to lead the search, *lb* (lower bound), Stop criterion, Test instance characteristics, and Performance measure. Therefore, the remaining columns contain the before-mentioned information for the four examined algorithms, Partial, RBS, NVST-IG+, and HTS, respectively. Next, we describe each of the analyzed characteristics.

- Approach: Name of the examined algorithm.
- Author: Name of the authors of the algorithm.
- Hybridization: Algorithms combined with another search strategy and the name of such search method.
- $R||C_{max}$ heuristics: This row remarks the approaches using $R||C_{max}$ problem domain strategies.
- Search strategies: Lists of the general search methods of each examined algorithm.
- Strategies to modify a solution: Methods used by the algorithms to generate, manipulate, and alter the solutions.
- Strategies to lead the search: Methods used to control how the solutions are modified.
- *lb*: Remarks the approaches using a lower bound to guide the search in the solution space and to fathom partial solutions that cannot lead to optimal ones.
- Stop criterion: Indicates the criterion used to finish the search.
- Test instance characteristics: Enumerates the general characteristics of the benchmark used to evaluate the performance of the examined algorithms.
- Performance measure: Depicts the measures used to estimate the performance of the algorithms.

Table 3.1: Main characteristics of the best algorithms of the state of the art of $R||C_{max}$.

Approach	**Partial [358]**	**RBS [374]**	**NVST-IG+ [346]**	**HTS [96]**
Author	Mokotoff and Jimeno	Ghirardi and Potts	Fanjul-Peyro and Ruiz	Sels et al.
Hybridization				B&B
$R\|\|C_{max}$ **Heuristics**	X	X	X	X
Search strategies	Partial solution Rounding phase	Constructive heuristic Filtering Node evaluation Recovering step	Constructive heuristic Variable neighborhood search Machine selection Jobs selection	Constructive hueristic Tabu list Branch and bound strategies
Strategies to modify a solution	B&B strategies	Pseudo-dominance conditions Truncated branch and bound	Insertion Interchange Restricted Local Search NSP and VIR	Swap Branch and bound strategies
Strategies to lead the search	R_1	R_1	R_2 R_3 R_4	R_1 R_2 R_3 R_4 R_5 R_6 R_7
lb	X	X		X
Stop criterion	60 seg		15 seg	15 seg
Test instance characteristics	p_{ij}: U(10, 100) m: from 3 to 20 n: from 10 to 50	p_{ij}: $U(10, 100)$, $U(10, 1000)$, and $MacsCorr$ m: from 10 to 50 n: from 100 to 1000	p_{ij}: $U(1, 100)$, $U(10, 100)$, $U(100, 120)$, $U(100, 200)$, $JobsCorr$, and $MacsCorr$ m: 10, 20, 30, 40, and 50 n: 100, 200, 500, and 1000	p_{ij}: $U(1, 100)$, $U(10, 100)$, $U(100, 120)$, $U(100, 200)$, $JobsCorr$, and $MacsCorr$ m: 10, 20, 30, 40, and 50 n: 100, 200, 500, and 1000
Performance measure	RPD to CPLEX	RPD to LB	RPD to CPLEX	RPD to CPLEX

Additionally, for a clearer format of Table 3.1, we named the strategies used to control the search as R_1, R_2, R_3, R_4, R_5, R_6, and R_7. In this way, the strategy R_1 controls the search process using a lower bound. On the other hand, the strategies R_2 and R_3 regulate the movements of the jobs. Thus, R_2 only accepts the operations (insertion and interchange) performed over the jobs when the resulted processing time C_i in the affected machines is equal to or lower than the current makespan, and R_3 only accepts a job movement if the sum of the processing time C_i in the affected machines will be better after the job movement. On the other hand, the strategy R_4 validates that from the intervened machines, one has low quality (with C_i close or equal to C_{max}) and the other one not. In the same order of ideas, the strategies R_5 and R_6 make sure that the selected jobs belong to distinct machines and that at least one of the chosen jobs has a different processing time on the other machine, respectively. Finally, the strategy R_7 validates that at least one of the intervened machines has a $C_i = C_{max}$.

From Table 3.1 can be seen that the only hybridized heuristic is HTS which incorporates a Branch and Bound (B&B) method. Likewise, it indicates that the Partial, RBS, and HTS algorithms share some characteristics, such as the fact that they use branching and bound strategies; therefore, they control the search process only using a lower bound. On the other hand, NVST-IG+ and HTS use similar heuristics to control how to modify the solutions, including the operations incorporated and the criteria used to accept or reject a job movement. Regarding the stop criterion, all the examined algorithms use time, except for RBS, which, being an exact algorithm, cannot stop until its search ends. Likewise, this table shows that the test instances used to evaluate the Partial

and RBS algorithm performance are different, while NVST-IG+ and HTS used the same benchmark with more varied characteristics. Finally, this table allows observing that the measure used to evaluate the performance of all the algorithms examined is the Relative Percentage Deviation RPD to CPLEX, except RBS, assessed with the RPD to a lower bound.

3.2 $R||C_{max}$ benchmark of instances

The general structure of a $R||C_{max}$ test instance can be defined using a matrix of size $(m \times n)$. Figure 3.2 shows a template of an $R||C_{max}$ instance. The $|M|$ columns from i_1 to i_m represent the available machines, and the rows indicate the $|N|$ jobs from j_1 to j_n to assign. Therefore, each cell in the matrix contains the processing time p_{ij} that machine i needs to process job j.

Machines

Jobs \ M	i_1	i_2	...	i_m
j_1	p_{11}	p_{21}	...	p_{m1}
j_2	p_{12}	p_{22}	...	p_{m2}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
j_n	p_{1n}	p_{2n}	...	p_{mn}

Figure 3.2: Template of a test instance of $R||C_{max}$.

As the review of the specialized literature related to $R||C_{max}$ presented in Section 3.1 indicates, in the first related works, each author generated their test instances (i.e., Martello *et al.* [354], Mokotoff and Jimeno [356, 358] or Ghirardi and Potts [374]). None of those instances are publicly available, nor their best-known solutions. Nevertheless, it is well-known that have been studied instances with p_{ij} values generated with a uniform distribution in intervals like $U(1, 100)$ and $U(10, 100)$, as well as with job and machine correlated ($JobsCorr$ and $MacsCorr$). That is, short (long) processing times p_{ij} of jobs for all machines and slow (fast) machines for all jobs, respectively.

It was until 2010 that Fanjul-Peyro and Ruiz introduced a collection of 1400 test instances available publicly at http://soa.iti.es and also used in this work [342]. This benchmark groups the 1400 instances into seven classes concerning the criteria employed to generate the processing times p_{ij} in each instance. Its characteristics are the following. The first five groups includes instances with values of p_{ij} uniformly distributed in the intervals $U(1,\ 100)$ $U(10,\ 100)$, $U(100,\ 120)$, $U(100,\ 200)$, and $U(1000,\ 1100)$, respectively; while the sixth and seventh groups include instances with correlated jobs ($JobsCorr$) and correlated machines ($MacsCorr$), respectively. Each class (sub-set) holds ten instances for each of the twenty possible combinations generated with $m = 10, 20, 30, 40, 50$ and $n = 100, 200, 500, 1000$. Thus, the size of the smaller instances is 100×10, and the size of the larger ones is 1000×50.

3.3 Analysis of the $R||C_{max}$ state-of-the-art algorithm results

In this section, we compare the results presented in the specialized literature for the best algorithms for the $R||C_{max}$ problem. For this review, we consider the best two-phase algorithm, Partial enumeration of Mokotoff and Jimeno [358]; the best exact method, Recovering Beam Search (RBS) of Ghirardi and Potts [374]; the best local search Iterative Greedy local search of Fanjul-Peyro and Ruiz, referred to as NVST-IG+ [342]; and the best hybrid method, the Hybrid Tabu Search of Sels et al., referred to as HTS [96]. Table 3.1 contains more details on the procedure for each solution method considered.

The results of the four state-of-the-art algorithm Partial algorithm, RBS, and NVST-IG+ are taken from the paper of Sels *et al.* [96]. In this work, they present a comparative performance of the four algorithms on solving the benchmark of 1,400 test instances introduced by Fanjul-Peyro in 2010, described in 3.2. This study is based on time, using a stopping criterion of 15 sec. However, given the characteristics of the Partial enumeration and the RBS procedures, they used more CPU time on average. It is important to note that they do not consider the remaining results presented by Fanjul-Peyro and Ruiz (2011) [342] because they are obtained by parallel algorithms run on a cluster of dual processors. Similarly, these results are not considered in this work, since they are outside the scope of study.

The performance of the state-of-the-art solution method is assessed based on the average Relative Percentage Deviation (RPD). Given an instance i, the RPD is defined as Equation 3.6, where $C_{max}(i)$ depicts the C_{max} value found by the assessed solution method and $C^*_{max}(i)$ represents the best C_{max} found using two hours of the commercial solver CPLEX. Thus, RPD indicates the deviation from the evaluated solution method to CPLEX.

$$RPD = \frac{C_{max}(i) - C^*_{max}(i)}{C^*_{max}(i)} \tag{3.6}$$

Table 3.2 contains the experimental results. This table presents the average RPD values reached by each solution method distributed in groups of instances sorted according to the distribution of their processing times p_{ij} and the complete benchmark (1400 instances). The first and second columns indicate the criteria used to group the test instances: p_{ij} and the 1400 instances. Therefore, the remaining columns contain the average RPD obtained by each approach for every group of instances, highlighting in bold the best values. It is important to note that the results presented in 3.2 are divided by 100 for the purposes of this research project.

Table 3.2: Comparison of the state-of-the-art algorithms Partial, RBS, NVST-IG+, and HTS using RPD.

Instance Set	**Partial**	**RBS**	**NVST-IG+**	**HTS**
$U(1, 100)$	0.0288	0.0203	**0.0134**	0.0183
$U(10, 100)$	0.0131	0.0187	**0.0075**	0.0151
$U(100, 120)$	0.0033	0.0013	0.0004	**0.0000**
$U(100, 200)$	0.0105	0.0081	0.0032	**0.0008**
$U(1000, 1100)$	0.0023	0.0018	0.0002	**-0.0001**
$JobsCorr$	0.0234	0.0035	0.0048	**-0.0053**
$MacsCorr$	0.0094	0.0236	0.0055	**0.0038**
1400 instances	0.0130	0.0110	0.0050	**0.0047**

From Table 3.2 emerged that the best state-of-the-art algorithm for $R||C_{max}$ is the HTS proposed by Sels *et al.* with an average RPD of 0.47 for the 1400 instances, followed by the NVST-IG+ of Fanjul-Peyro and Ruiz with an average deviation to CPLEX of 0.50. In contrast, the heuristics with the lowest performance were the Partial algorithm of Mokotoff and Jimeno and the RBS of Ghirardi and Potts with RPD values of 1.30 and 1.10, respectively. Finally, Table 3.2 allows observing that the sets of instances that seem to be the most difficult are $U(1, 100)$ and $U(10, 100)$ since they represent a greater difficulty for state-of-the-art algorithms, being NVST-IG+ the algorithm with the best results. On the other hand, HTS presents the best results in the rest of the sets. However, we consider that this comparison is not appropriate since HTS was programmed and run in a programming language and computer equipment with different characteristics, respectively, from the remaining assessed algorithms. In this order of ideas, we believe that using the number of evaluations of the objective function is the fairest way to compare the algorithms' performance. Thus, we can avoid the execution time impact caused by the programming language and the computer equipment characteristics. Finally, we consider that only comparing the algorithms' performance based on the average RPD for the 1400 instances or the instance sets grouped according to the processing times distribution is not suitable since not all the instances have the same weight. Therefore, in the following sections, we incorporate other criteria such as the number of machines m and the number of jobs n in the instances to compare the algorithmic behavior of the proposed strategies more fairly. From these observations arise the following research questions: (1) What will make the difference in the difficulty of the instances and the behavior of the algorithms?; (2) Is it possible to design a competitive GGA for $R||C_{max}$ by incorporating knowledge of the problem domain? The following chapters present a series of experimental studies that seek to answer such research questions.

Chapter 4

The first GGA to solve the $R||C_{max}$ problem

This chapter presents the first GGA for the NP-hard combinatorial optimization grouping problem $R||C_{max}$. In this way, it includes an overview of the GGA procedure, highlighting its main features and strengths over other heuristic algorithms. In addition, this section describes the heuristics of each GGA component, including the population initialization strategy, the crossover and mutation operators, and the reproduction technique. Finally, it includes the experimental results obtained by GGA when solving the $R||C_{max}$ problem.

According to Ramos-Figueroa *et al.* [7], the Grouping Genetic Algorithm (GGA) is one of the most used metaheuristic algorithms to solve grouping problems. Such popularity is related to its promising results and its flexibility to adopt new ideas to handle the constraints and conditions of the problem to solve. The GGA is an extension to the standard GA; then, they have similar procedures. The GGA procedure starts with the generation of the initial population, generally in a random way. Next, selection strategies and variation operators, mainly crossover and mutation, are used iteratively to try to find better solutions. Each iteration represents a generation that starts utilizing a selection strategy to pick some individuals of the populations, based on their fitness values. Therefore, the genetic material of the selected individuals is recombined by the crossover operator to generate offspring. Subsequently, the offspring are added to the population using a replacement strategy. Finally, some individuals, chosen with a selection strategy, are slightly modified with the mutation operator. In this way, the GGA iterates performing the before-mentioned procedure until a stopping criterion (e.g., the maximum number of generations, the maximum search time, convergence of solutions, or finding an optimal solution) is met.

One of the main features of the GGA is the group-based scheme that it uses to encode and manage solutions in the search space. According to Falkenauer, this is a more natural way of representing solutions to grouping problems [93]. Besides, it helps to reduce the size of the search space since it produces fewer isomorphic solutions than a traditional encoding. In this encoding, each gene represents a group that contains the

collection of elements that correspond to it. Therefore, the length of a solution is equal to the number of groups that it includes.

Another important aspect to consider when developing a GGA is the design of variation operators, like crossover and mutation, that must work at the group-level. With this feature, operators can perform procedures in a more controlled way, determining which groups and elements vary according to the constraints and objectives of the problem to solve. The crossover uses two or more solutions of the current population to recombine their genetic material, creating offspring with new characteristics. This operator is used to give GGA the ability to converge on the most promising areas identified during the search. One of the advantages of crossover operators for group-based encoding is that they can use the quality of the groups to determine how the parents transmit the genetic material to the children to perform a more controlled search. On the other hand, the mutation operator provides to GGA the ability to explore new areas of the search space, producing small modifications to the genetic material of some solutions. This procedure is helpful for a GGA, mainly to address highly constrained grouping problems, where there are large possibilities of converging to local optimums since these slight alterations generate solutions in other regions of the search space that can help to avoid premature convergence.

The next sections describe the elements of the first GGA for $R||C_{max}$, the object of study in this work [7], including the population initialization strategy, the variation operators, selection and replacement strategies, as well as the problem-domain heuristics. It is important to note that this algorithm is an adaptation of the state-of-the-art Grouping Genetic Algorithm with Controlled Genes Transmission (GGA-CGT) introduced by Quiroz-Castellanos *et al.* to solve the Bin Packing problem [11]. Therefore, the details of the original heuristic procedures can be consulted in such work.

4.1 Genetic encoding, fitness function and initial population

The proposed GGA uses the group-based representation scheme to encode and manipulate solutions, where each machine i is a gene (or group) G_i that will include a set of jobs. Therefore, all solutions have the same number of genes, equal to the number of machines m. The quality of each machine i is equal to the time it takes to process its assigned jobs, denoted as C_i. Thus, the quality of a solution C_{max} is equal to the C_i value of the machine with the longest processing time. The initial population is generated randomly by running the Random min strategy that consists of applying the well-known Min() heuristic, introduced by Ibarra and Kim [346], on random jobs permutations. Recalling from Chapter 3, for each job j, Min() calculates the equation C_i=$C_i + p_{ij}$ for all the machines, where p_{ij} indicates the time that machine i needs to process job j. In this way, Min() assigns job j to machine i that generates the lowest C_i value.

Figure 4.1 describes the procedure followed by the population initialization strategy. To give a comprehensive description, Figure 4.1a includes an example instance I

represented as a matrix with m=4 machines depicted by the columns and n=10 jobs represented by the rows. Thus, the example starts from a permutation (Figure 4.1b) of the ten jobs $\{j_9, j_5, j_2, j_6, j_3, j_8, j_4, j_7, j_1, j_{10}\}$, used to generate the partial solution, shown in Figure 4.1c. The construction of the partial solution can be calculated from the first nine jobs in the permutation $\{j_9, j_5, j_2, j_6, j_3, j_8, j_4, j_7, j_1\}$ and the instance I using the heuristic Min(). To exemplify how this heuristic Min() works, Figure 4.1d shows a complete solution, resulted from the assignment of the last job in the permuted list (i.e., j_{10}) to the solution. Therefore, following the Min() procedure, the processing time C_i of each machine plus the time that they require to process job j_{10} results in the following way: $C_1 = 26 + 8$, $C_2 = 25 + 20$, $C_3 = 20 + 18$, and $C_4 = 10 + 28$. In this manner, Min() assigned job j_{10} to machine i_1 since it generated the lowest C_i value. It is important to note that if two or more machines produce the same C_i value, this allocation heuristic assigns the job in turn to the machine i that appears first from i_1 to i_m. Therefore, Figure 4.1d also represents a solution chromosome with a length equal to the number of machines m, where each machine represents a gene. Finally, Figure 4.1d also shows the fitness value of the generated solution that is equal to the longest processing time C_i, in this case, the C_1=34, outlined in bold.

a) Test Instance

M \ N	i_1	i_2	i_3	i_4
j_1	16	15	6	25
j_2	10	10	16	18
j_3	18	15	6	10
j_4	15	10	16	20
j_5	29	20	5	19
j_6	20	12	5	16
j_7	11	19	3	3
j_8	12	10	10	20
j_9	28	20	12	7
j_{10}	8	20	18	28

b) Permutation: $j_9, j_5, j_2, j_6, j_3, j_8, j_4, j_7, j_1, j_{10}$

c) Partial solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_2	j_3, j_4	j_5, j_6, j_8	j_7, j_9
C_i	26	25	20	10

d) Complete solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_2, j_{10}	j_3, j_4	j_5, j_6, j_8	j_7, j_9
C_i	**34**	25	20	10
	C_{max}			

Figure 4.1: Population initialization strategy

4.2 Adapted Gene-level crossover operator

The GGA uses the Adapted Gene-Level Crossover (AGLX) operator, a variant of the GLX operator proposed by Quiroz-Castellanos *et al.* [11], that produces two offspring by using two parents. During the genetic material transmission process, this operator considers the genes in increasing order concerning the C_i values. Thus, it transmits first the machines that process their jobs fastest and then the slowest ones. In this way, the first child starts inheriting the fastest machine from the first parent, next the fastest machine from the second parent, then the second-fastest machine from the first

parent, and so on. It is important to note that, before transmitting each machine i, this crossover operator verifies that it has not already been transmitted by the other parent. Otherwise, the machine is discarded. Similarly, the second child receives genes alternately from both parents, but it starts with the fastest machine from the second parent. Finally, to avoid infeasible solutions, this crossover operator removes the jobs that appear twice from the machine with the higher C_i value. Finally, AGLX re-inserts the jobs missed during the transmission process using the assignment heuristic Min().

Figure 4.2 describes the process of the AGLX operator with an example that contains two parent solutions for the test instance of Figure 4.1a with four machines (groups). The ten jobs, from j_1 to j_{10}, are distributed among the four machines, from i_1 to i_4, and the time that each machine i requires to process its assigned jobs from C_1 to C_4 is stored in vector C_i. Figure 4.2a depicts the transmission process. Therefore, it shows the two parents with their groups in increasing order, which indicates the gene transmission sequence, i.e., from best (Lowest C_i) to worst (Highest C_i). Figure 4.2b indicates the way the repeated genetic material is handled. Thus, it contains the two solutions produced during the transmission process, which only keep machine i of the parent in which it appears first according to the gene transmission sequence. Furthermore, this figure includes the repeated jobs, highlighted in bold, that must be removed from the machine with the highest processing time C_i. Lastly, this figure shows a list with the missed jobs (MJ) during the transmission process. Figure 4.2c contains the partial solution resulting from the transmission process without the repeating genetic material, as well as a permutation of the jobs in MJ. Finally, Figure 4.2d shows the complete solutions resulting from the assignment of the missed jobs with the heuristic Min().

4.3 Download mutation operator

The GGA includes the Download mutation operator that uses two phases to modify two genes in each solution. In the first stage, called download, the operator clusters the genes (machines) between two sets (W and O). In this way, W includes the machines with a processing time (C_i) equal to the makespan (C_{max}), while O holds the ones with an assigned processing time (C_i) lower than the makespan (C_{max}). Next, from each set (W and O), one machine (w and o) is randomly selected, and their jobs are released. Subsequently, in the second stage, the released jobs are redistributed between the selected machines (w and o) with the heuristic Best(). For each job j, Best() calculates the equations $C_w = C_w + p_{wj}$ and $C_o = C_o + p_{oj}$, where C_w and C_o represent the assigned processing time of machines w and o, respectively, and p_{wj} and p_{oj} the processing time required by machines w and o to process job j. In this way, Best() assigns j to the machine that generates the lowest C_i value. The main difference between the reassignment heuristics Min() and Best() is that Min() re-inserts the jobs considering all the machines, while Best() re-inserts them by considering only the two selected machines o and w.

Figure 4.3 describes the mutation process of the Download operator with an example that contains an initial solution for the instance presented in Figure 4.1a with four genes (groups). The ten jobs, from j_1 to j_{10}, are distributed among four groups, from

Given two parent solutions for the test instance of Figure 1a, the Adapted Gene-level crossover operator (AGLX) proposed by Ramos-Figueroa *et al.* [2] works as follows:

a) Transmission process	Machines	i_4	i_3	i_2	i_1		
	Jobs	j_7, j_9	j_5, j_6, j_8	j_3, j_4	j_1, j_2, j_{10}	First Parent	
	C_i	10	20	25	34		
	Machines	i_2	i_3	i_1	i_4		
	Jobs	j_2	j_1, j_3, j_6, j_7	j_5, j_9	j_4, j_8, j_{10}	Second Parent	
	C_i	10	20	57	68		
b) Repeated genetic material	Machines	i_4	i_2	i_3	i_1		
	Jobs	j_7, j_9	j_2	j_5, j_6, j_8	$\mathbf{j_5, j_9}$	First Child	*MJ* j_1, j_3, j_4, j_{10}
	C_i	10	10	20	57		
	Machines	i_2	i_4	i_3	i_1		
	Jobs	j_2	j_7, j_9	$j_1, j_3, j_6, \mathbf{j_7}$	$j_5, \mathbf{j_9}$	Second Child	*MJ* j_4, j_8, j_{10}
	C_i	10	10	20	57		
c) Partial solution	Machines	i_1	i_2	i_3	i_4		
	Jobs		j_2	j_5, j_6, j_8	j_7, j_9	First Child	Permutation j_{10}, j_4, j_3, j_1
	C_i	0	10	20	10		
	Machines	i_1	i_2	i_3	i_4		
	Jobs	j_5	j_2	j_1, j_3, j_6	j_7, j_9	Second Child	Permutation j_8, j_{10}, j_4
	C_i	29	10	17	10		
d) Offspring	Machines	i_1	i_2	i_3	i_4		
	Jobs	j_1, j_{10}	j_2, j_4	j_5, j_6, j_8	j_3, j_7, j_9	First Child	
	C_i	24	20	20	20		
	Machines	i_1	i_2	i_3	i_4		
	Jobs	j_5	j_2, j_4, j_8	j_1, j_3, j_6, j_{10}	j_7, j_9	Second Child	
	C_i	29	30	17	10		

Figure 4.2: Adapted Gene-Level Crossover (AGLX) operator

i_1 to i_4, and the time that each group i requires to process its assigned jobs from C_1 to C_4 is stored in vector C_i. Figure 4.3a shows the result of clustering the machines with processing time C_i equal to the makespan C_{max} in the set W=$\{i_1\}$ and the remaining machines in set O=$\{i_2,\ i_3,\ i_4\}$. Figure 4.3b indicates the machines w=i_1 and o=i_4, outlined in bold, randomly selected from the sets W and O, respectively. Figure 4.3c contains the solution with the selected machines to be altered, outlined in bold, downloaded by releasing their jobs and placing them in the box of released jobs RJ. Finally, Figure 4.3d has a permutation of the jobs in RJ and the result of reinserting them with the allocation heuristic Best(). The calculation of the processing time C_i of each machine i, as well as the operations performed by the allocation heuristic Best() to assign the released jobs, can be calculated by using the example instance I presented in Figure 4.1a. As this example indicates, the quality of the mutated solution is better than that of the initial solution, demonstrating the effectiveness of the Download mutation operator.

Given the following potential solution for the test instance of Figure 1a:

Solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_2, j_{10}	j_3, j_4	j_5, j_6, j_8	j_7, j_9
C_i	34	25	20	10

The Download mutation operator proposed by Ramos-Figueroa *et al.* [2] works as follows:

a) Machines in the sets *W* and *O*

i_1 — W

i_2, i_3, i_4 — O

b) Selecting machines *w* and *o*

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_2, j_{10}	j_3, j_4	j_5, j_6, j_8	j_7, j_9
C_i	34	25	20	10
	w			o

c) Download

Machines	i_1	i_2	i_3	i_4
Jobs		j_3, j_4	j_5, j_6, j_8	
C_i	0	25	20	0

RJ: $j_1, j_2, j_7, j_9, j_{10}$

d) Reinsertion

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_{10}	j_3, j_4	j_5, j_6, j_8	j_2, j_7, j_9
C_i	24	25	20	28

Permutation: $j_1, j_{10}, j_2, j_9, j_7$

Figure 4.3: Download mutation operator

4.4 Selection and replacement strategies

The GGA employs an adaptation of the controlled reproduction technique proposed by Quiroz-Castellanos *et al.* [11], which uses an elitist approach together with two inverted rankings to give all the solutions a chance to contribute to the next generation but forcing the survival of the best solutions. The replacement strategy preserves the population diversity and the best solutions by replacing duplicated fitness individuals and the worst fitness solutions with new offspring.

At each generation, GGA ranks the individuals in the population P from best to worst according to their fitness. Additionally, if there are solutions with repeated fitness, only one solution is ranked, and the others are placed at the end of the ordered list. Subsequently, GGA distributes the solutions in P, ranked according to the ordered list among the sets G, R, and B. The set G includes the best n_c solutions, where n_c is a parameter to be configured that determines the number of individuals selected for the crossover process at each generation. On the other hand, the set R contains the solutions in the population P without the best $n_c/2$ solutions. Finally, the set B holds the best $|B|$ individuals, called elite solutions, that receive special treatment since they have the best characteristics of the population. Therefore, $|B|$ is another parameter to be configured. Given this hierarchical structure of the solution, $n_c/2$ parent solutions are randomly taken from the set G and the remaining $n_c/2$ parents are randomly picked up from the solutions in the set R. In this way, each pair of parents is created with a

parent selected from the set G and the other one from the set R. Hence, it is necessary to validate that parent pairs do not have the same solution since some solutions can be selected more than once. After applying the crossover operator to each pair of parents, the new individuals are incorporated into the population P in the following way. Half of the generated children replaces the parents selected from the set R and the remaining offspring replaces first the solutions with repeated fitness and then those with worse fitness. The detail of this reproduction technique can be consulted in [11].

Once the replacement strategy is applied, the population is ranked with the same ranking strategy, i.e., from best to worst and placing solutions with repeated fitness at the end, to later select the best n_m solutions for mutation, where n_m is a parameter to be configured that determines the number of mutated solutions at each generation. When applying the mutation operator, if a solution belongs to the elite group B, the solution is first cloned, and it is later mutated. The clones can be entered into the population, replacing first the solutions with repeated fitness and then those with worse fitness. This strategy aims to preserve the individuals with the best characteristics to take advantage of the search directions they provide for a greater number of generations. A more detailed description of this reproduction technique can be found in [11].

4.5 Computational Experiments

This section presents the experimental design proposed to analyze the performance of the GGA presented in this section to solve the $R||C_{max}$ problem, an adaptation of the state-of-the-art GGA-CGT designed to solve the Bin Packing problem [11]. It is important to note that, for this study, the heuristic used to generate the initial population in GGA-CGT, as well as the mutation and crossover operators, were adapted to solve the problem $R||C_{max}$. Conversely, the remaining mechanisms and operators, as well as the parameter settings, were not modified. The configuration used is as follows: Population size $|P| = 100$; number of individuals selected for the crossover $n_c = 20$; number of individuals selected for the mutation $n_m = 83$; elite population size $|B| = 20$; and, maximal number of generations $max\_gen = 500$.

The performance assessment of GGA involves solving the benchmark of 1,400 test instances introduced by Fanjul-Peyro in 2010, described in 3.2. Likewise, we analyze its performance by measuring its average Relative Percentage Deviation (RPD) to CPLEX, presented in Equation 3.6. Table 4.1 contains the experimental results. For a comprehensive analysis, we distributed the RPD values reached by the GGAs in groups of instances sorted according to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij} of the instances, and the complete benchmark (1400 instances). The first and second columns indicate the criteria used to group the test instances: n, m, p_{ij}, and the 1400 instances. On the other hand, the third column contains the average RPD obtained by GGA for the four grouping criteria. It is important to note that from this study, we will use these extended tables to analyze in-deep the algorithmic behavior of the algorithms and the conditions in which they show high or low performance. This table format represents the first step toward the characterization of the $R||C_{max}$ problem optimization process since it

Table 4.1: Analysis of the average RPD reached by GGA for each instance set: n, m, p_{ij}, and the 1400 instances.

Instance Set		GGA
n	100	0.0659
	200	0.0655
	500	0.0657
	1000	0.0688
m	10	0.0683
	20	0.0683
	30	0.0683
	40	0.0683
	50	0.0683
p_{ij}	$U(1, 100)$	0.1027
	$U(10, 100)$	0.1119
	$U(100, 120)$	0.0256
	$U(100, 200)$	0.0829
	$U(1000, 1100)$	0.0121
	$JobsCorr$	0.0586
	$MacsCorr$	0.0955
1400 Instances		0.0699

provides meaningful information to understand how the characteristics of the instances, like the number of machines m and the number of jobs n, impact the performance of the algorithms that solve them.

From Table 4.1 can be observed that GGA did not show an outstanding performance since the average RPD obtained for the state-of-the-art HTS for the 1400 instances (See Table 3.2) is about fourteen times lower than the RPD obtained by the proposed GGA. Additionally, Table 4.1 suggests that the GGA performance improves as the number of jobs decreases; while the number of machines does not have a clear impact. Finally, these experimental results suggest that, like in the state-of-the-art algorithms, instances with processing times in the ranges $U(1, 100)$, $U(10, 100)$, and with correlated machines $MacsCorre$ represent a bigger challenge for the GGA; while instances in the ranges $U(100, 120)$ and $U(1000, 1100)$ are easier.

4.6 Impact analysis of crossover and mutation rate on GGA

This section presents an experimental study to analyze how each variation operator (crossover and mutation) impacts the GGA performance. In this sense, we conducted a set of tests that considers three different values for the number of individuals selected for the crossover process (n_c) and the number of solutions to be mutated (n_m), both

with the following values: 20, 40, and 60. In this way, we run the GGA with the nine configurations ($Conf$) generated from all possible combinations of these three parameters: $Conf_1$: $n_c = 20$, $n_m = 20$, $Conf_2$: $n_c = 20$, $n_m = 40$, ... $Conf_9$: $n_c = 60$, $n_m = 60$. Figure 4.4 has a bar graph with the results obtained from this study, where each bar represents one of the nine configurations grouped according to the number of mutated solutions n_m, and each pattern indicates the number of selected individuals for the crossover process n_c: squares = 20, waves= 40, and circles= 60. As Figure 4.4 indicates, the GGA performance tends to improve (lower RPD) as the number of individuals considered for the crossover and mutation processes increases. Moreover, this figure suggests that the crossover operator shows a higher impact on the GGA performance. This behavior is different from the one presented by the GGA-CGT, where the mutation operator has the most significant positive impact on the final performance of this algorithm. The conclusions obtained from this study suggest the review and re-structuration of the mutation operator procedure. Chapter 7 presents a systematical experimental design to explore different strategies that intervene during the mutation process. Thus, we will identify and select the strategies that positively contribute to the mutation operator procedure to improve its performance.

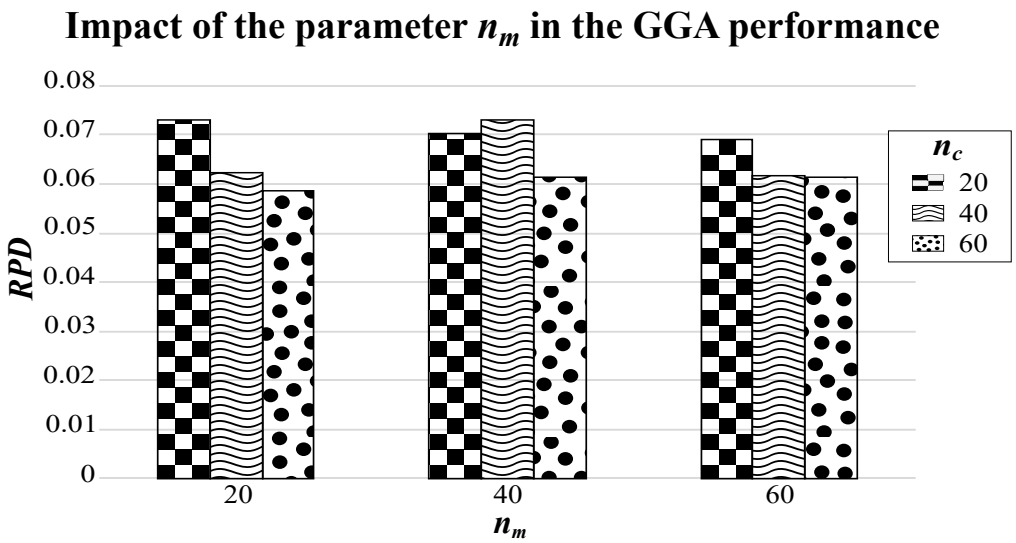


Figure 4.4: Impact analysis of the parameters: number of individuals selected for crossover n_c and number of mutated solutions n_m in the GGA final performance.

4.7 Conclusions of the experimental study

In this chapter, we introduced the first GGA to solve the Parallel-Machine Scheduling variant that considers unrelated machines, jobs with non-preemptions, and makespan minimization ($R||C_{max}$). Our proposal was designed based on the trends identified during the literature review. That is, using a more natural or intuitive way to represent the potential solutions to the $R||C_{max}$ problem (group-based) as well as variation operators designed to work together with the grouping encoding efficiently.

Experimental results indicated that, although the proposed GGA showed promising results for $R||C_{max}$, these are not enough to compete with state-of-the-art algorithms. In this order of ideas, the best state-of-the-art algorithm, HTS, is about fourteen times better than GGA since they solve the 1400 instances with average RPD values of 0.0047 and 0.0699, respectively (see Table 3.2). Moreover, the impact analysis of the parameters revealed that the mutation operator is not contributing much to the search. The knowledge gained from this research phase motivates the development of the following sections, where we will analyze if it is possible to improve the GGA performance without the need to add a local search as the best state-of-the-art heuristics do. Therefore, in the following sections, we will explore different heuristic strategies, seeking to identify the appropriate ones for each stage of the evolutionary process.

Chapter 5

Population initialization strategies

This chapter addresses the study of different heuristics to construct solutions for the Parallel-machine scheduling problem with unrelated machines and makespan minimization $R||C_{max}$. Recalling from Chapter 3, $R||C_{max}$ holds hard complexity; consequently, the state-of-the-art includes methods designed to solve it under different approaches, such as deterministic heuristics [345, 346, 347, 348], exact methods [96, 354, 356, 358, 374], two-phase algorithms or rounding methods [343, 349, 361, 364, 370, 371, 372, 357], local searches [352, 96, 352, 353, 359, 352, 359, 373, 367, 353, 373, 351, 342, 365], evolutionary algorithms[96, 352, 375], and hybrid algorithms [352, 96]. As with most search algorithms, the starting point for a GGA is the construction of the initial population. Many times this component is not given the necessary importance. However, its role is very relevant, since, in some problems, the quality and other characteristics of the initial solutions have a great impact on the final performance of a solution method. In this chapter, we analyze the algorithmic behavior of eleven simple heuristics to generate solutions, five deterministic, and six non-deterministic. The knowledge gained from this study is used as a starting point for the characterization of the $R||C_{max}$ problem and the optimization process of the GGA components.

5.1 State-of-the-art constructive heuristics for the $R||C_{max}$ problem

According to the scope of the literature review, the earliest efforts to solve $R||C_{max}$ were devoted to the design of simple solution methods, mainly constructive heuristics. The first work related to the design of constructive heuristics was presented in 1977 by Ibarra *et al.*. In this work, the authors introduced five well-known heuristics that use different criteria to establish the scheduling order of the jobs, but they share in common that they use the allocation heuristic Min() to control the scheduling of each job. From that work, we consider three heuristics, referred in this study as Lowest min, Highest min, and Random min. Three years later, De and Morton studied the strengths and weaknesses of the heuristics proposed by Ibarra *et al.*; as a result, they introduced three constructive algorithms that incorporate new strategies to establish the way the

jobs are assigned to the machines, including a new strategy to sort the jobs and a lower bound [347]. In this chapter, we also consider one of these heuristic algorithms, referred to as Mean min. Finally, the last constructive heuristic found in the specialized literature that we also use in this work is Lowest, introduced by Fanjul-Peyro and Ruiz in 2010 to generate the initial solution of different neighborhood searches [342]. As can be seen, the state-of-the-art of constructive heuristics for $R||C_{max}$ is not too large. However, some of them are still widely used for the construction of the initial solutions of different methods. The following sections present a study to analyze the performance of the heuristic strategies of the state-of-the-art indicated in this review, together with other heuristics proposed in this work.

5.2 Constructive heuristics for the $R||C_{max}$ problem

In deterministic algorithms, given a particular input, the same output is always produced, passing through the same sequence of states. Algorithms with these characteristics are far functional so that they can be run on real devices (e.g., machines, computers, robots, etc.) efficiently. This study comprises four classical deterministic algorithms: Lowest [342], Lowest min [346], Highest min [346], Mean min [347], and one deterministic heuristic proposed in this work, referred to as Diff_fastest min. In contrast, non-deterministic algorithms can generate different outputs on different runs, even for the same input. In most cases, such behavior is related to the use of random number generators. Algorithms with these characteristics are far recurrent to address complex problems, with large and hard search spaces. In this work, we analyze the algorithmic behavior of six non-deterministic constructive heuristics, one taken from the specialized literature known as Random min [346], and five more introduced in this work, called: Random, Random lowest bound min, Lowest 4g min, Highest 4g min, and Diff_fastest 4g min. Most of them use the scheduling strategy Min() heuristic, introduced by Ibarra and Kim [346]. Recalling from Chapter 3, the heuristic Min() allocates each job j to the machine i that produces the lowest value of $C_i = C_i + p_{ij}$, where C_i is the time demanded by the machine i to process all its assigned jobs, and p_{ij} is the time demanded by machine i to process job j. However, some of these constructive heuristics differ in the strategy they use to sort the jobs. Figure 5.1 contains an instance example with $n = 8$ jobs and $m = 4$ machines, as well as the detail of four properties used to classify the jobs before schedule them: 1) *lowest*, the processing time p_{ij} required by the fastest machine i to process job j; 2) *highest*, the processing time p_{ij} required by the slowest machine i to process job j; 3) *mean*, the average processing time required by all the machines in $M = \{i_1, ..., i_m\}$ to process job j; and 4) *diff_fastest*, the difference between the two fastest machines to process job j. The following paragraphs contain the description of the eleven constructive heuristics considered in this chapter.

N \ M	i_1	i_2	i_3	i_m	*lowest*	*highest*	*mean*	*diff_fastest*
j_1	20	15	10	12	10	20	14.25	2
j_2	10	12	16	18	10	18	14	2
j_3	18	15	11	10	10	18	13.5	1
j_4	15	17	16	11	11	17	14.75	4
j_5	10	20	11	19	10	20	15	1
j_6	20	12	15	16	12	20	15.75	3
j_7	11	19	20	13	11	20	15.75	2
j_n	12	10	18	20	10	20	15	2

Figure 5.1: Instance characteristics used by constructive heuristics.

5.2.1 Lowest

This method is the simplest since it does not use a strategy to arrange the jobs. Instead, it schedules the jobs directly considering the property *lowest*, described in Equation 5.1, without any other criteria. As Figure 5.1 indicates, this property refers to the minimum time $\min(p_{ij})$ required to process the job j.

$$lowest\ p_{ij} = min(p_j) \tag{5.1}$$

5.2.2 Lowest min

Unlike Lowest, the heuristic Lowest min uses instance property *lowest*, described in Equation 5.1, to sort the jobs in non-increasing order. Thus, the heuristic Lowest min allocates the sorted jobs with the Min() heuristic.

5.2.3 Highest min

Similar to Lowest min, Highest min allocates the jobs in non-increasing order, but based on the property *highest*, using the Min() heuristic. As can be seen from Figure 5.1 and Equation 5.2, the property *highest* refers to the maximum processing time $max(p_{ij})$ required by the slowest machine i to process job j.

$$highest\ p_{ij} = max(p_j) \tag{5.2}$$

5.2.4 Mean min

In this algorithm, jobs are scheduled employing the Min() heuristic, in non-increasing order, according to the instance property *mean*. From Figure 5.1 and Equation 5.3 can be seen that this characteristic refers to the average processing time $average(p_{ij})$ of each job j.

$$mean\ p_{ij} = average(p_j) \tag{5.3}$$

5.2.5 Diff_fastest min

Finally, we propose a deterministic heuristic called: Diff_fastest min. This algorithm considers the jobs in non-increasing order according to the instance property *diff_fastest*, described in Equation 5.4, to schedule them using the Min() heuristic. Figure 5.1 shows the result of extracting the characteristic *diff_fastest* from an instance example. As can be seen, it consists of identifying the two machines (i_a and i_b) that require the lowest time p_{ij} to process job j, to then calculate the absolute value of the difference between the processing times of these machines $abs(p_{i_aj} - p_{i_bj})$.

$$diff\_fastest\ p_{ij} = abs(p_{i_aj}\text{-}p_{i_bj}) \tag{5.4}$$

5.2.6 Random

In this non-deterministic heuristic, jobs are allocated randomly on machines without considering other criteria. Thus, this heuristic uses the function random() that generates a number randomly with a uniform distribution between 1 and the number of machines m to determine the location of each job. The main benefit of the Random heuristic relies on its applicability to problems that need too much diversity since it can generate solutions with different characteristics. In contrast, effectiveness is the principal drawback of this heuristic since it cannot control the quality of the solutions. Therefore, it generates high-quality and low-quality solutions.

5.2.7 Random min

Unlike Random, the Random min heuristic combines randomness and knowledge of the problem domain to generate the solutions as follows. First, it selects each job using the function $random()$ that generates a random number with a uniform distribution between 1 and the number of jobs n. Then, it allocates the selected job with the well-known Min() heuristic. In this way, the Random min heuristic iteratively performs this procedure until all the jobs are assigned.

5.2.8 Random lowest bound min

The Random lowest min bound heuristic is an extension to the Random min heuristic that incorporates the use of the lower bound lb. As Equation 5.5 indicates, lb consists of summing the processing times p_{ij} required by the fastest machines to process every job j and dividing the result by the number of available machines m. In this order of ideas, the Random lowest min bound procedure is as follows. For each job j selected with the random() function that uses a uniform distribution, this strategy attempts to assign j to machine i that processes it in the lowest time p_{ij}, i.e., using the *lowest* property, considering the following rule. If $C_i + p_{ij} <= lb$, job j is assigned to the fastest machine i. Otherwise, job j remains unassigned. C_i represents the time that machine i needs to process its assigned jobs. Finally, the Random lowest min bound heuristic allocates the unassigned jobs with the Min() heuristic.

$$lb = \frac{\sum_{j=1}^{n} min(p_{ij})}{n} \tag{5.5}$$

5.2.9 Lowest 4g min

This heuristic is an extension of the deterministic constructive heuristic Lowest min. Therefore, the constructive process of Lowest 4g min also starts ordering the jobs in non-increasing order based on the instance characteristic *lowest*, described in Equation 5.1; nevertheless, it does not assign the jobs immediately. Instead, it distributes them into four groups, from G_1 to G_4 with the same number of jobs equal to $n/4$, preserving the order of the jobs during group construction. Subsequently, the jobs in each group are permuted, modifying their order. Finally, the jobs are allocated with the Min() heuristic, following the order of the groups: G_1, G_2, G_3, and G_4.

5.2.10 Highest 4g min

Like Lowest 4g min, Highest 4g min is an extension of the deterministic constructive heuristic Highest min. Hence, it starts arranging the jobs in non-increasing order based on the instance characteristic *highest*, described in Equation 5.2. Next, it distributes the jobs into four groups, from G_1 to G_4 in equal parts of size $n/4$, preserving their order during group construction. Subsequently, the jobs of each group are permuted with a uniform distribution, changing their order. Finally, the jobs are allocated with the Min() heuristic following the order of the groups: G_1, G_2, G_3, and G_4.

5.2.11 Diff_fastest 4g min

Besides the heuristics Lowest 4g min and Highest 4g min, we also used the approach based on four groups with the instance property *diff_fastest*, described in Equation 5.4. As can be inferred, in this heuristic, jobs are first arranged in non-increasing order based on the subtraction of the processing time p_{ij} required by the two fastest machines (i_a and i_b) for processing each job j. Next, the Diff_fastest 4g min process continues as that of the heuristics Lowest 4g min and Highest 4g min. That is, distributing the jobs into four groups, permuting them, and finally assigning them with the Min() heuristic.

5.3 Analysis of the $R||C_{max}$ constructive heuristics results

This section includes the experimental results for the eleven studied constructive heuristics. The experimental design consists of evaluating the final performance of the eleven constructive heuristics described above to understand their optimization process when solving $R||C_{max}$. To achieve this goal, we proposed an experimental design that consists of assessing the eleven constructive heuristics as follows. First,

each constructive heuristic is applied to the 1400 instances. Next, the performance of each constructive heuristic is calculated based on the measure RPD, presented in Equation 3.6. Finally, for a comprehensive analysis, the performance of the eleven constructive heuristics is compared with the 1400 instances grouped with four criteria, the number of jobs n, the number of machines m, the distribution of the processing times p_{ij}, and the 1400 instances together. It is important to note that, for a fair comparison, the same seed is used for the eleven constructive heuristics.

Table 5.1 presents the experimental results. The first two columns indicate the criteria used to group the instances, i.e., n, m, p_{ij}, and the complete benchmark. Thus, the remaining columns contain the average RPD obtained by each constructive heuristic for each grouping criterion, highlighting in bold the best results.

Table 5.1: Comparison of the eleven constructive heuristics: Lowest, Lowest min, Highest min, Mean min, Diff_fastest min, Random, Random min, Random lowest bound min, Lowest 4g min, Highest 4g min, and Diff_fastest 4g min using RPD.

Instance Set		Lowest	Lowest min	Highest min	Mean min	Diff_fastest min	Random	Random min	Random lowest bound min	Lowest 4g min	Highest 4g min	Diff_fastest 4g min
n	100	0.549	0.186	0.172	0.184	0.176	0.493	0.171	0.337	**0.132**	0.148	0.167
	200	0.515	0.173	0.167	0.176	0.174	0.478	0.165	0.284	**0.134**	0.151	0.164
	500	0.45	0.142	0.138	0.143	0.15	0.464	0.139	0.221	**0.12**	0.132	0.141
	1000	0.41	0.133	0.13	0.134	0.144	0.456	0.131	0.187	**0.12**	0.127	0.134
m	10	0.326	0.136	0.14	0.146	0.153	0.412	0.14	0.129	**0.124**	0.135	0.144
	20	0.44	0.158	0.155	0.162	0.164	0.462	0.154	0.225	**0.131**	0.145	0.155
	30	0.491	0.151	0.142	0.152	0.149	0.487	0.142	0.268	**0.115**	0.129	0.14
	40	0.549	0.164	0.149	0.16	0.158	0.5	0.15	0.312	**0.122**	0.135	0.15
	50	0.601	0.183	0.173	0.177	0.18	0.504	0.171	0.352	**0.14**	0.154	0.169
p_{ij}	$U(1,100)$	0.406	0.3	0.307	0.326	0.313	0.919	0.303	0.647	**0.196**	0.299	0.3
	$U(10,100)$	0.38	0.243	0.223	0.244	0.238	0.801	0.222	0.425	**0.202**	0.219	0.217
	$U(100,120)$	0.512	0.061	0.06	0.063	0.061	0.114	0.06	**0.054**	0.06	0.06	0.06
	$U(100,200)$	0.377	0.139	0.136	0.142	0.138	0.373	0.136	0.136	**0.134**	0.136	**0.134**
	$U(1000,1100)$	0.387	0.034	0.035	0.035	0.034	0.059	0.034	**0.023**	0.034	0.034	0.034
	Jobscorre	0.541	0.148	0.148	0.15	0.152	0.337	0.153	0.138	**0.076**	**0.076**	0.153
	Macscorre	0.765	0.184	**0.153**	0.155	0.19	0.708	**0.153**	0.377	0.183	**0.153**	0.163
1400 instances		0.481	0.158	0.152	0.159	0.160	0.473	0.151	0.257	**0.126**	0.139	0.151

Block one of Table 5.1 suggests that the number of jobs n in an instance gives some information related to its difficulty. As can be seen in this block, most constructive heuristics reached a better average RPD in the instances with the greatest number of jobs (n=1000), while as the number of jobs n decreases, their RPD values grow. To in-depth analyze this behavior, we generated box plot graphs to analyze how the

number of jobs n impacts the algorithmic behavior of the eleven constructive heuristics. Figures 5.2 and 5.3 contain the generated box plot graphs for the deterministic and non-deterministic heuristics, respectively. The $x-$axis indicates the number of jobs n, and the $y-$axis depicts the RPD. The box plots in Figures 5.2 and 5.3 allow reiterating that from the eleven heuristics studied, only the random heuristic does not present the behavior mentioned above (if the number of jobs n decreases, the difficulty of an instance also increases).

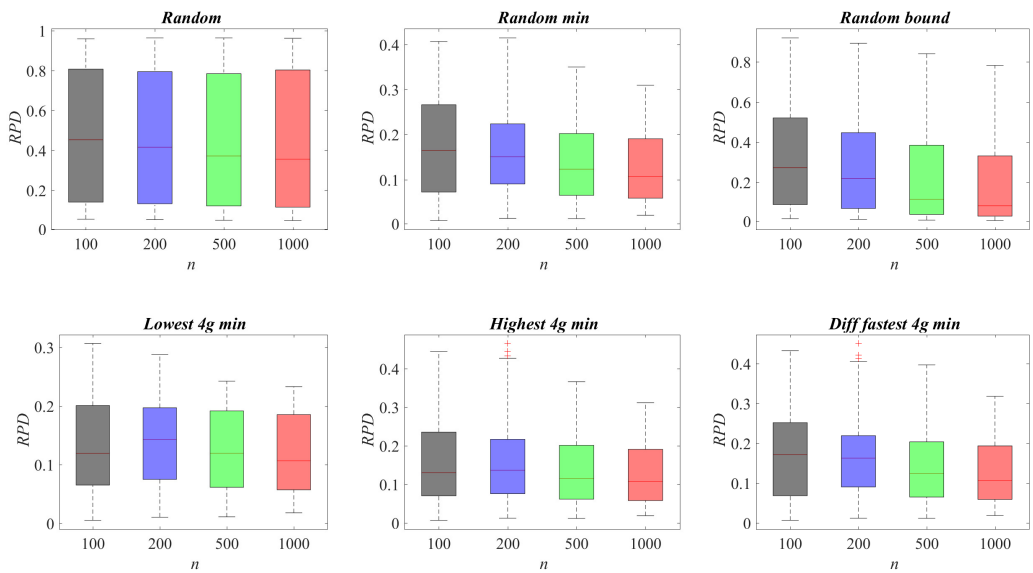


Figure 5.2: Performance of deterministic heuristics for instances grouped by number of jobs n.

On the other hand, the second block of Table 5.1 suggests that the number of machines m has a contrary impact on the difficulty of the instances. That is, most constructive heuristics get the best average RPD by solving instances with the lowest number of machines (m=10), and such average increases as the number of machines m grows. To analyze this behavior, we generated box plot graphs that show how the number of machines m influences the optimization process of the eleven constructive heuristics. Figures 5.4 and 5.5 contain the generated box plot graphs for the deterministic and non-deterministic heuristics, respectively. The $x-$axis indicates the number of machines m, and the $y-$axis depicts the RPD. From these plots can be observed that the eleven heuristics studied present the above-mentioned behavior. However, it is less marked in heuristics like Lowest 4g min, Highest 4g min, and Diff_fastest 4g min.

Similarly, from the third block of Table 5.1 can be inferred that the constructive heuristics generate solutions with a lower RPD when solving instances where the quotient q of the maximum $max(p_{ij})$ by the minimum $min(p_{ij})$ processing time is less. Figure 5.6 indicates the average quotient of the instances of each set concerning the distribution of the processing times. As can be seen in this plot, the set $U(1000, 1100)$ contains the instances with the lowest q, while the ones with the highest q are in the set $U(1, 100)$. To analyze this behavior in deep, we created 2D dispersion graphs

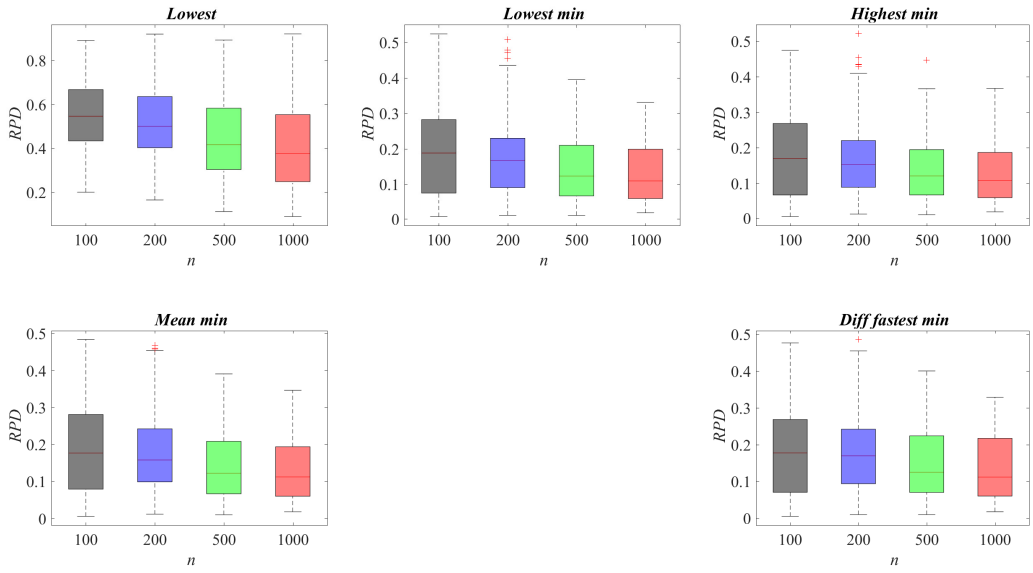


Figure 5.3: Performance of non-deterministic heuristics for instances grouped by number of jobs n.

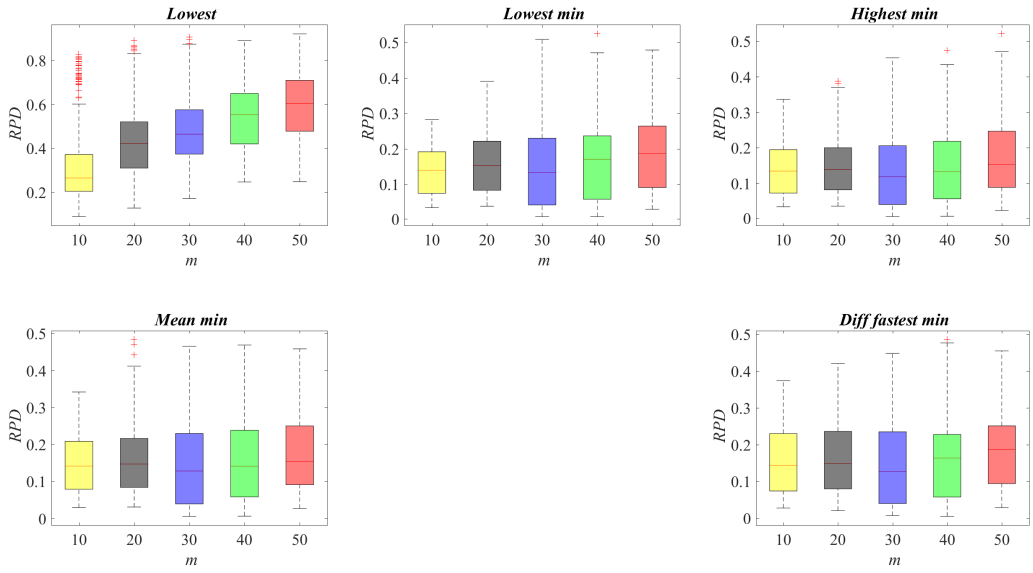


Figure 5.4: Performance of deterministic heuristics for instances grouped by number of machines m.

that show how the quotient q of the instances impacts the optimization process of the eleven constructive heuristics. Figures 5.7 and 5.8 contain the generated dispersion graphs for the deterministic and non-deterministic heuristics, respectively. The $x-$axis indicates the quotient q of the instances, and the $y-$axis depicts the RPD. From Figure 5.7 can be seen that the deterministic constructive heuristics reach better results in

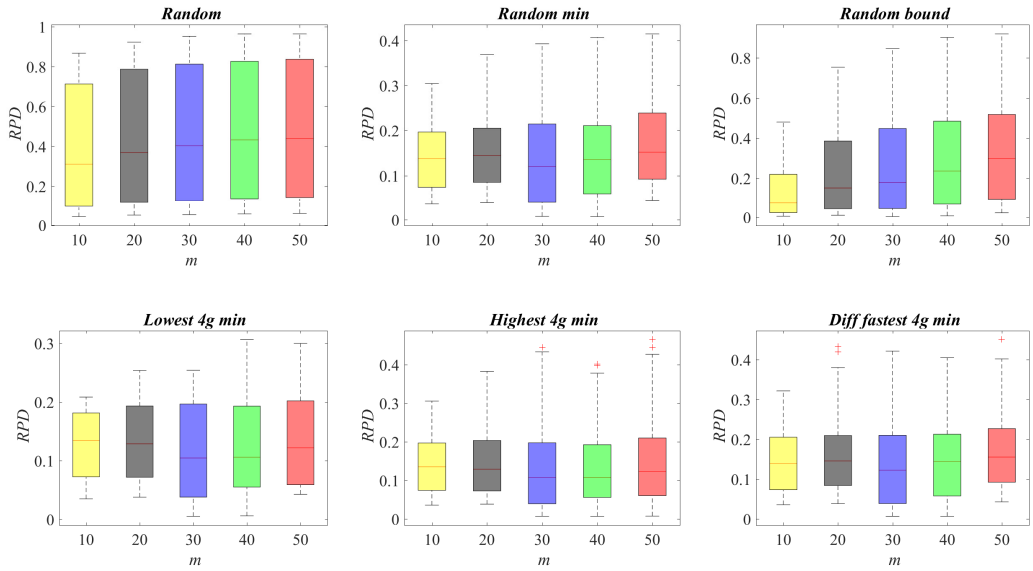


Figure 5.5: Performance of non-deterministic heuristics for instances grouped by number of machines m.

instances where the processing time distribution is between 1000 and 1100 (i.e., where $q = 1.1$). A possible reason is that the processing times p_{ij} are quite similar since the highest processing time is only 1.1 times greater than the lowest one. Besides, these plots show how the performance of constructive heuristics decreases as the value of q increases. That is, the heuristics reach the worst performance in instances where the processing time distribution is from 1 to 100 since the highest processing time can be 100 times greater than the lowest one. Similarly, Figure 5.8 suggests that q impacts on the performance of non-deterministic constructive heuristics, such that the higher the value of q, the greater the difficulty of the instance.

Finally, the last block of Table 5.1 indicates that the constructive heuristic with the best performance is Lowest 4g min, followed by the other two variants that used the proposed approach that arrange the jobs based on four groups. Highest 4g min and Diff_fastest 4g min. It is important to note that although Lowest 4g min showed the best performance, we are going to keep the Random min heuristic as the population initialization strategy of the GGA since it allows generating good solutions with greater diversity.

5.4 Conclusions of the analysis

In this chapter, we presented a study to analyze the algorithmic behavior of eleven constructive heuristics, five taken from the state-of-the-art and six more introduced in this work. We evaluated the performance of each constructive heuristic on solving the 1400 $R||C_{max}$ test instances. Subsequently, we conducted an investigation, based on exploratory data analysis techniques, looking for the characteristics that make difficult

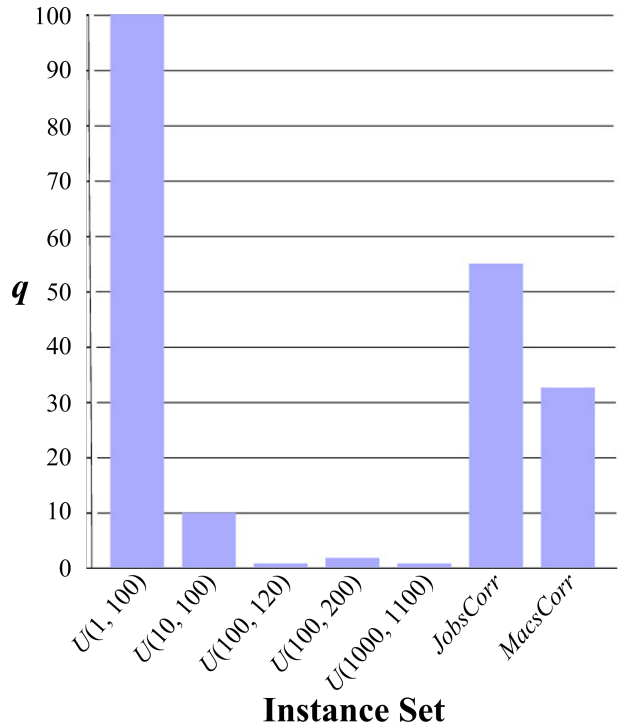


Figure 5.6: Average quotient q of the instances in each set, grouped according to the criterion used to generate the values of p_{ij}.

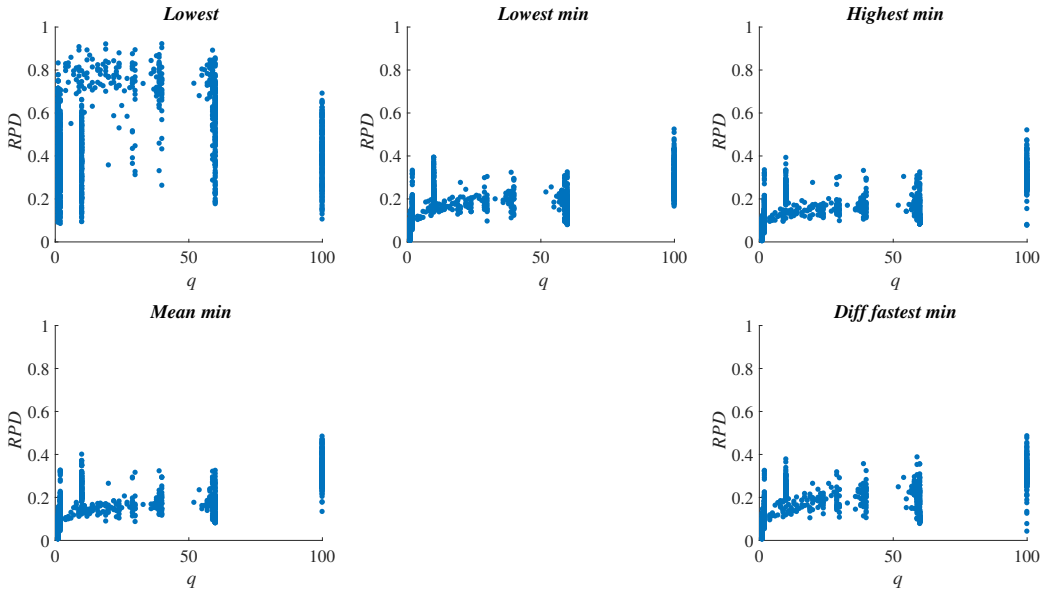


Figure 5.7: Performance of deterministic heuristics with references to the quotient of the maximum by the minimum processing time (q) of the instances.

an $R||C_{max}$ instance for a constructive heuristic. The graphical and tabular analysis allowed observing how the number of machines m and jobs n of the instances, as well

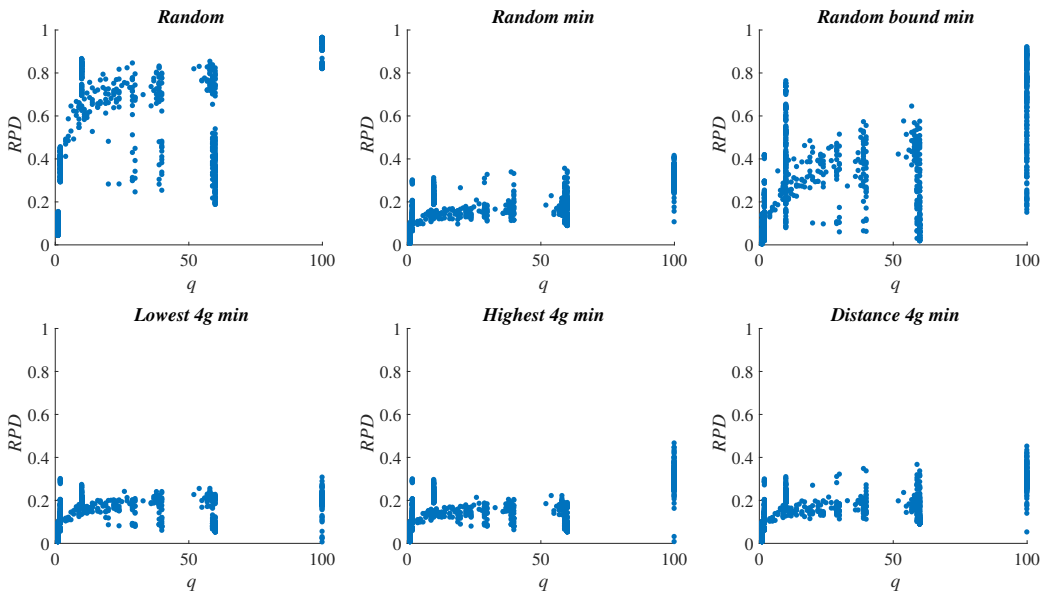


Figure 5.8: Performance of non-deterministic heuristics with references to the quotient of the maximum by the minimum processing time (q) of the instances.

as the quotient q of their processing times p_{ij}, can impact the performance of the constructive heuristics. In this way, we observed that an instance represents a higher challenge for a constructive heuristic as the number of jobs n decreases, the number of machines m grows, and the quotients q increases. This study allowed us to have the first step towards the characterization of the $R||C_{max}$ problem and the optimization process of its solution methods, which will be taken up in the following chapters. Concerning the overall performance of the constructive heuristics, the experimental results revealed that the constructive heuristic with the best results is Lowest 4g min. However, we will use the Random min constructive heuristic as the GGA population initialization strategy since it is simple and generates solutions close to the best known with different characteristics. This behavior is important because it will allow the GGA to start with good quality solutions without leading to premature convergence in most of the test instances. In the following chapters, we will analyze the optimization process of the rest of the GGA components in order to create a specific-purpose GGA for $R||C_{max}$, making use of the knowledge gained from each study conducted.

Chapter 6

Crossover operators

One of the features that differentiate the genetic algorithms from other search methods is the crossover operator, used to recombine the genetic material of two or more solutions in order to create solutions with new characteristics. The performance of a crossover operator can vary according to the properties of the problem to solve. Therefore, it is important to perform an experimental study to identify the operator that best suits to the search space of the addressed problem. To facilitate this goal, it is essential to know the state-of-the-art crossover operators designed to solve grouping problems, the special class of combinatorial optimization problems to which belongs the $R||C_{max}$ problem, since their procedures can be used as a design guide.

This chapter presents an experimental study to characterize the algorithmic behavior of crossover operators for $R||C_{max}$. The study covers a review of the state-of-the-art crossover operators for grouping problems, highlighting their main heuristic strategies and their applicability in grouping problems with different characteristics. Additionally, this section includes an examination of the optimization process of the most outstanding crossover operators in the specialized literature to solve grouping problems, adapted to solve $R||C_{max}$. Finally, this section presents an experimental design to analyze how the different aspects that intervene during the crossover process impact on its final performance, like the way the machines are arranged in parents and how they are transmitted to offspring, the number of generated children, and the repeated genetic material handling. In this way, we analyze the algorithmic behavior of thirty-two crossover operators, including the state-of-the-art operators and the crossover operators generated from the systematical experimental study. The information gathered gives an overview of the options that exist. Therefore, it will be used as inspiration to design a specific-purpose crossover operator for $R||C_{max}$. The designed crossover operator was incorporated into the GGA presented in Chapter 4, giving rise to the first Enhanced GGA (EGGA). Experimental results indicated that the EGGA performance had a slight improvement rate of about 17%.

6.1 State-of-the-art of grouping crossover operators

The crossover operator is the most used variation operator (operators applied to the population to generate solutions with new characteristics) to design GGAs since all the GGAs designed to solve grouping problems incorporate this operator. Therefore, at present, there are several grouping crossover operator variants. The importance of this operator lies on its procedure since it is in charge of establishing the search directions according to the different constraints and conditions of the problem to solve. During this research stage, we conducted a detailed review of the state-of-the-art crossover operators for GGAs that solve grouping problems. We found out that there was a gap in the specialized literature since there are crossover operators that have different names but use the same procedure to recombine the genetic material of the solutions. In the same way, we found cases where two crossover operators with different names perform the same procedure. The foregoing may be related to the fact that the properties and restrictions of each problem make it necessary to adjust the procedure of the crossover operators to avoid generating infeasible solutions, which resulted in those confusions.

The above motivated the meticulous analysis of each state-of-the-art crossover operator designed to solve grouping problems, which allowed classifying them by making a generalization of their procedure. We proposed three taxonomies, called variation-degree, solutions encoding, and parameter setting-level. The details of the compiled information and the proposed taxonomies can be found in [376].

Tables 6.1, 6.2, and 6.3 present the variation-degree taxonomy. The columns of these tables hold general information about the crossover operators found in the state-of-the-art, including their names, abbreviations used in this work, and references to related works. This taxonomy is based on the variation-degree (i.e., the way in which parents transmit the genetic material to their children, whether based on segments, groups, or items) that the crossover operators cause to the solutions. Thus, it sorts them according to three criteria, called segment-oriented, group-oriented, and item-oriented crossover operators.

In accordance with this study, the state-of-the-art contains five segment-oriented crossover operators, so-called 1PX (One-point Crossover), 2PX (Two-points Crossover), 3PX (Three-points Crossover), 4PX (Four-points Crossover), and MPX (Multi-point Crossover). Usually, these operators use two solutions (sol_1 and sol_2) of the current population as parents (P_1 and P_2) to generate one (C) or two (C_1 and C_2) children (offspring). The general procedure of segment-oriented crossover operators is randomly picking out k crossing points to divide parent solutions into $k+1$ segments of genes, which defines x crossing segments (cs). After this segmentation, different strategies determine the transmission order of the cs created. It is important to note that the genetic material transmission processes used by the five segment-oriented crossover operators can generate infeasible solutions. Therefore, it is necessary to incorporate problem-domain heuristics to transform them into feasible ones. Table 6.1 contains the general information of the five segment-oriented crossover operators found in the state-of-the-art.

Table 6.1: Segment-oriented crossover operators. *Operator*: Name of the operator. *Abbr.*: Abbreviation of the operator name. *References*: Related works.

Operator	***Abbr.***	***References***
One-point	1PX	[22, 29, 15, 54, 55, 56, 45, 21, 39]
Two-points	2PX	[23, 24, 25, 26, 27, 28, 8, 10, 12, 45, 61, 62, 51, 89, 41, 88, 72, 49, 76, 43, 73, 74, 60, 53, 66, 67, 33, 34, 37, 36, 35, 92, 70, 90, 377, 30, 378, 379, 380, 381, 71, 46, 86, 20, 48, 83, 38, 18]
Three-points	3PX	[44, 77, 78, 79, 80, 91, 81, 82]
Four-points	4PX	[378, 379, 47]
Multi-points	MPX	[31]

Unlike the operators that work at the segment-level, most group-oriented crossover operators perform the transmission of the genetic material considering the particular characteristics of every group, according to the used encoding. This behavior is remarkable since it allows performing a more controlled mating process. According to this review, to date, the state-of-the-art contains eight variation operators designed with this approach. Table 6.2 includes the general information of the eight group-oriented crossover operators found in the state-of-the-art.

Table 6.2: Group-oriented crossover operators. *Operator*: Name of the operator. *Abbr.*: Abbreviation of the operator name. *References*: Related works.

Operator	***Abbr.***	***References***
Uniform	UX	[15, 64, 85]
Exon shuffling	ESX	[10, 14, 17]
Gene-level	GLX	[11, 16, 7, 84, 87]
Greddy partition	GPX	[13, 63, 52, 69, 65, 59, 19, 75, 50, 32]
Lowest index max	LIMX	[15, 52]
Lowest index first	LIFX	[52]
Multi-chromosomal	MX	[9]
Distance preserving	DPX	[68]

Finally, the state-of-the-art also holds link-oriented crossover operators, designed to work directly with the links of the linear linkage encoding. According to this review, the specialized literature only includes two link-oriented operators: UX (Uniform Crossover) and MUX (Modified uniform Crossover). Table 6.3 shows the general information of the two link-oriented crossover operators found in the state-of-the-art.

Table 6.3: Link-oriented crossover operators. *Operator*: Name of the operator. *Abbr.*: Abbreviation of the operator name. *References*: Related works.

Operator	***Abbr.***	***References***
Uniform	UX	[15]

Modified uniform	MUX	[15]

It is important to note that the related works considered in this literature review include GGAs with different representation schemes. Therefore, not all the identified crossover operators apply to this work. The following section presents an experimental study of state-of-the-art crossover operators that have shown promising results and that apply for the GGA with the group-based representation scheme.

6.2 Experimental design for the $R||C_{max}$ crossover operators

This section includes the experimental design proposed to analyze how the aspects that intervene during the recombination process of the crossover operators impact their performance. The main objective of this study is to identify the heuristic strategies that positively impact the performance of grouping crossover operators on solving the problem $R||C_{max}$, to design an efficient grouping crossover that improves the performance of the GGA presented in Chapter 4. The experimental design consists of five phases that consider: (1) the performance analysis of the state-of-the-art crossover operators, (2) strategies to sort the genes of parent solutions, (3) heuristics to establish the gene transmission order and the number of children, (4) strategies to handle the repeated genetic material, and (5) a study to analyze the utility of the proposed operator. It is important to note that, to establish the order used to analyze the crossover operator aspects, we gave priority to the general elements such as the rearrangement and the process of gene transmission. Thus, we leave the more fine factors for the end, such as repeated genetic material.

The performance evaluation of each crossover operator is conducted as follows. First, a population of 100 individuals is randomly generated with the selected constructive heuristic from the previous chapter, i.e. Random min. Next, the assessed crossover operator is applied to the population for 500 generations. In each generation, the 100 solutions are used as parents to generate offspring, forming pairs of parents randomly. In this way, the children of each generation replace their parents, i.e., we use the generational replacement.

Subsequently, the operators are compared as follows. First, each crossover operator is applied to the 1400 instances. Later, the performance of each crossover operator is calculated based on the measure RPD, presented in Equation 3.6. Finally, for a comprehensive analysis, the performance of the crossover operators is compared with the 1400 instances grouped with four criteria: the number of jobs n, the number of machines m, the distribution of the processing times p_{ij}, and the 1400 instances together. It is important to note that, for a fair comparison, the same seed is used for all the crossover operators. The experimental results are presented in tables, where the first two columns indicate the criteria used to group the instances, i.e., n, m, p_{ij}, and the complete benchmark. Thus, the remaining columns contain the average RPD

obtained by each crossover operator for each grouping criterion, highlighting in bold the best results.

The following sections describe the procedure of the operators studied in each phase, contain a comparison of their performance, and highlight the characteristics that show a positive impact on solving the $R||C_{max}$ problem.

6.2.1 State-of-the-art operators

Before starting with the design of the new operator, we analyzed the performance of some state-of-the-art crossover operators to get an overview of what could be achieved with existing operators. The literature review indicates that the most used and best-performing approaches have been the group-based operators since they control the genetic material transmission process, taking advantage of the characteristics and properties of the groups. Given the above, this section only considers group-based operators for the group-based representation scheme, covering Exon Shuffle Crossover (ESX), Gene-Level Crossover (GLX), Greedy Partition Crossover (GPX), and Uniform Crossover (UX). For a fair comparison, during the crossover processes, the machines with at least one repeated job are eliminated, as this is the most common approach to handle the repeated genetic material based on the scope of the literature review. The specific procedure of each operator adapted to solve the problem $R||C_{max}$ is described below.

The ESX operator uses two parents P_1 and P_2 to generate a child C. Their crossover process begins by joining the machines of the two parents, to later sort them from best to worst based on the processing time that they need to process their assigned jobs C_i (i.e., from lowest to highest C_i). As a result, a list of ordered machines ML is created, where each machine is twice, once for each parent. In this way, the child is constructed as follows. The child C begins by receiving the best of all machines. Then, the second machine is reviewed, which is transmitted as long as it is not repeated, and it does have no repeated jobs (i.e., a job that is part of one of the previously transmitted machines). Generally, this process ends with infeasible solutions with some empty machines and some missed jobs. Said solutions are transformed into feasible ones using the Min() assignment heuristic, which places each missed job into the machine that affects the solution's makespan to a lesser extent.

Unlike ESX, the GLX operator uses two parents P_1 and P_2 to generate two children C_1 and C_2. Its crossover process begins by sorting both parents' machines from best to worst based on the time they need to process their assigned jobs C_i (that is, from lowest to highest C_i). In this way, children are created considering the arranged machines of the two parents in parallel. That is, first, the best machine of the first parent P_1 is compared with the best machine of the second parent P_2; then, the comparison is performed with the second-best machine of each parent, and so on. In this way, the process is repeated to compare all the machines in parallel. During the genetic material transmission process, GLX can take two paths. First, if the compared machines have different values of C_i, both children C_1 and C_2 receive first the machine with the lowest C_i and then the other one. On the other hand, if both machines have the same C_i

vales, the first child C_1 receives first the machine of the first parent P_1 and then the machine of the second parent P_2, while the second child C_2 first receives the machine from the second parent P_2 and then the machine from the first parent P_1. This process of transmitting genetic material can also end with incomplete solutions, with some empty machines and some missed jobs. Therefore, like in the ESX operator, the missed jobs are re-inserted using the Min() allocation heuristic.

Like GLX, the GPX operator uses two parents P_1 and P_2 to generate two children C_1 and C_2, its crossover process begins by classifying the machines of both parents from best to worst based on the processing time that they need to process their assigned jobs, and consider the ordered machines of the two parents in parallel. That is, it first considers the best machine from the first parent P_1 and the best machine from the second parent P_2, then it considers the second-best machine from each parent, and so on. The main difference between GLX and GPX lies in their gene transmission process, since GPX uses two probability vectors, one for building each child. Thus, to generate the first child C_1, GPX uses a probability p generated with a uniform distribution for each pair of arranged machines. If $p \leq 0.5$, C_1 receives the machine from the first parent P_1. Otherwise, C_1 receives the machine from the second parent P_2. It is important to note that before transmitting each machine, GPX verifies that it is not repeated and that it does not have repeated jobs. Otherwise, the machine is discarded. The second child C_2 is created using the same process, but with a different vector of probabilities. Like ESX and GLX, GPX produces infeasible solutions. Therefore, it uses the Min() allocation heuristic to re-insert jobs missed during the genetic material transmission process.

Finally, the UX operator also uses two parents P_1 and P_2 to generate two children C_1 and C_2. This is one of the simplest since it does not use any type of bias to control the crossover process. UX considers the machines of the two parents P_1 and P_1 always in the same order, i.e., from i_1, to i_m. Thus, for each pair of machines (one for each parent), it generates a random probability p with a uniform distribution. If $p \leq 0.5$, the first child C_1 receives the machine from the first parent P_1 and the second child C_2 receives the machine from the second parent P_2. Otherwise, the roles are inverted. Therefore, the first child C_1 receives the machine from the second parent P_2 and the second child C_2 receives the machine from the first parent P_1. The solutions resulting from this process may be feasible. Therefore, it is necessary to use the Min() allocation heuristic to transform infeasible solutions into feasible ones. Table 8.1 shows the RPD values reached from the four state-of-the-art crossover operators considered to CPLEX.

As can be observed in Table 8.1, the crossover operator with the best performance on solving the problem $R||C_{max}$ is ESX, which discards the randomness to exploit a bit of information about the solutions, i.e., the quality of the groups (the time that each machine i requires to process its assigned jobs C_i). Table 8.1 suggests that ESX reaches the best performance, even grouping the instances with the four different criteria considered. However, this table also highlights that the performance of the operators GLX, GPX, and UX is very close, and that all of them are quite far from CPLEX. The information gained from this study validates the necessity of developing a crossover operator with knowledge of the $R||C_{max}$ problem-domain, that reaches high-quality results. The following sections include the proposed operators for each aspect of the

Table 6.4: Comparison of the crossover operators: ESX (Exon Shuffle Crossover), GLX (Gene-Level Crossover), GPX (Greedy Partition Crossover) and UX (Uniform Crossover) using RPD.

Instance		**ESX**	**GLX**	**GPX**	**UX**
n	100	**0.155**	0.170	0.204	0.220
	200	**0.142**	0.167	0.198	0.206
	500	**0.132**	0.157	0.187	0.186
	1000	**0.153**	0.181	0.210	0.234
m	10	**0.153**	0.180	0.215	0.232
	20	**0.152**	0.178	0.214	0.229
	30	**0.152**	0.176	0.213	0.226
	40	**0.152**	0.175	0.211	0.223
	50	**0.153**	0.173	0.210	0.221
p_{ij}	$U(1, 100)$	**0.277**	0.398	0.476	0.606
	$U(10, 100)$	**0.160**	0.292	0.306	0.385
	$U(100, 120)$	0.070	**0.051**	0.073	0.054
	$U(100, 200)$	**0.162**	0.161	0.185	0.182
	$U(1000, 1100)$	0.038	**0.026**	0.038	0.027
	$jobsCorre$	0.210	0.157	0.181	**0.155**
	$MachCorre$	**0.147**	0.176	0.249	0.206
1400 instances		**0.152**	0.180	0.216	0.231

recombination process studied, and the results gained from each of them.

6.2.2 Strategies to rank the machines

One of the main benefits of group-level crossover operators is their capacity to distinguish between groups according to their quality. This particularity has motivated its implementation in several problems, showing promising results, which can be observed in the related works listed in Tables 6.1, 6.2, and 6.3. Given this premise and the knowledge gained from the experimental study of the state-of-the-art crossover operator performance, this and the following phases of the study are focused on identifying different aspects that intervene during the recombination process of grouping crossover operators. Thus, we will analyze how these aspects impact on the algorithmic behavior of the recombination process of a group-based crossover operator, to identify those that positively impact on its final performance. As a consequence, the gained knowledge will be used as a guide to design a specific-purpose crossover operator for $R||C_{max}$.

The design of the new grouping crossover operator starts from a simple and mostly random initial operator, called Random Grouping Crossover (RGX), to avoid all kinds of bias. Like most crossover operators, RGX generates two children C_1 and C_2 by

recombining the genetic material of two parents P_1 and P_2, as follows. First, it performs a permutation of the machines of each parent, to later transmit the machines to the children alternately, considering the arrangements generated by the permutations. That is, the first child C_1 receives the first machine from the first parent P_1, then the first machine from the second parent P_2, later the second machine from the first parent P_1, and so on, until it collects all machines from both parents. The second child C_2 is created similarly, but its creation process starts with the first machine of the second parent P_2, then the first one from the first parent P_1, and so on. It is important to note that duplicated machines are discarded, i.e., when RGX tries to transmit a machine from parent to child, it first verifies that the child has not yet received that machine from the other parent. Furthermore, before transmitting a machine, RGX validates that none of its jobs has already been transmitted with another machine. Otherwise, the machine is also discarded. Finally, the jobs missed during the transmission process are re-inserted using the assignment heuristic Min().

Figure 6.1 shows the recombination process followed by the RGX operator to generate two children from two parents. In the example, each parent has five machines and the machines have one or more jobs assigned. Additionally, each solution has a vector C_i, used to indicate the time that each machine i requires to process its assigned jobs. The processing time C_i of each machine can be calculated with the example instance I, where each column represents a machine i from i_1 to i_5 and each row depicts a job j from j_1 to j_{10}. In this way, each cell contains the processing time p_{ij} that the machine i requires to process the job j. Thus, Figure 6.1a shows the result of applying a random permutation to the order of the machines of each parent. Figure 6.1b contains the result of transmitting the permuted machines from parents to children alternately, discarding the repeated genetic material. Figure 6.1c includes the generated offspring with the repeated jobs that lead to removing a machine highlighted in red. Furthermore, this figure contains the missed jobs MJ during the genes transmission process of each child. Figure 6.1d presents the partial solutions (children) with the five machines and their respective assigned jobs, as well as a permutation of the missed jobs MJ of each child. Finally, Figure 6.1e shows the final solutions (children) resulting from the assignment of the missed jobs MJ to the partial solutions using the Min() allocation heuristic.

The first aspect to analyze about the crossover process is the arrangement that the machines have in the parents before conducting the genetic material transmission. In this order of ideas, this section presents different strategies to organize the solution machines to study the profits of considering the machines in a different order from i_1, i_2,...,i_m. In this fashion, this phase of the experimental study focuses on exploring different criteria to organize the machines in parents. We analyze the performance of six strategies to rank the machines, called Permutation (the one of RGX), Average(p_i), N_{jobs}, C_i, N_{jobs} - C_i, and C_i-N_{jobs}.

Figure 6.2 describes the process of each strategy with an example that contains two parent solutions for the test instance I with five machines (groups) and ten jobs. Thus, each parent contains the ten jobs, from j_1 to j_{10}, distributed among the five machines, from i_1 to i_5, and the time that each machine i requires for processing its assigned jobs from C_1 to C_5 is stored in vector C_i. Thus, Figure 6.2a shows the two parent solutions with their machines ranked with a permutation generated with a uniform

Given two parent solutions, and a test instance I:

Machines	i_1	i_2	i_3	i_5	i_4	
Jobs	j_1	j_3, j_4	j_5, j_6, j_8	j_7, j_9	j_2, j_{10}	First Parent
C_i	16	25	20	50	46	

Machines	i_1	i_2	i_3	i_4	i_5	
Jobs	j_4, j_8, j_{10}	j_1, j_3	j_2, j_9	j_5	j_6, j_7	Second Parent
C_i	35	30	28	19	29	

Test Instance I

m \ n	i_1	i_2	i_3	i_4	i_5
j_1	16	15	6	25	5
j_2	10	10	16	18	8
j_3	18	15	6	10	1
j_4	15	10	16	20	12
j_5	29	20	5	19	29
j_6	20	12	5	16	6
j_7	11	19	3	3	23
j_8	12	10	10	20	10
j_9	28	20	12	7	27
j_{10}	8	20	18	28	2

The Random Grouping Crossover (RGX) operator works as follows:

Permutation

Machines	i_2	i_4	i_1	i_3	i_5	
Jobs	j_3, j_4	j_2, j_{10}	j_1	j_5, j_6, j_8	j_7, j_9	First Parent
C_i	25	46	16	20	50	

Machines	i_5	i_3	i_4	i_1	i_2	
Jobs	j_6, j_7	j_2, j_9	j_5	j_4, j_8, j_{10}	j_1, j_3	Second Parent
C_i	29	28	19	35	30	

Genetic material transmision

Machines	i_2	i_5	i_4	i_3	i_1	
Jobs	j_3, j_4	j_6, j_7	j_2, j_{10}	j_2, j_9	j_1	First Child
C_i	25	29	46	28	16	

Machines	i_5	i_2	i_3	i_4	i_1	
Jobs	j_6, j_7	j_3, j_4	j_2, j_9	j_2, j_{10}	j_1	Second Child
C_i	29	25	28	46	16	

Repeated genetic material

Machines	i_2	i_5	i_4	i_3	i_1		
Jobs	j_3, j_4	j_6, j_7	j_2, j_{10}	~~j_2, j_9~~	j_1	First Child	*MI* j_5, j_8, j_9
C_i	25	29	46	~~28~~	16		

Machines	i_5	i_2	i_3	i_4	i_1		
Jobs	j_6, j_7	j_3, j_4	j_2, j_9	~~j_2, j_{10}~~	j_1	Second Child	*MI* j_8, j_{10}
C_i	29	25	28	~~46~~	16		

Partial solution

Machines	i_1	i_2	i_3	i_4	i_5		
Jobs	j_1	j_3, j_4		j_2, j_{10}	j_6, j_7	First Child	Permutation j_9, j_5, j_8
C_i	16	25	0	46	29		

Machines	i_1	i_2	i_3	i_4	i_5		
Jobs	j_1	j_3, j_4	j_2, j_9	j_5	j_6, j_7	Second Child	Permutation j_{10}, j_8
C_i	16	25	28	19	29		

Offspring

Machines	i_1	i_2	i_3	i_4	i_5	
Jobs	j_1	j_3, j_4	j_3, j_8, j_9	j_2, j_{10}	j_6, j_7	First Child
C_i	16	25	27	46	29	

Machines	i_1	i_2	i_3	i_4	i_5	
Jobs	j_1, j_{10}	j_3, j_4, j_{10}	j_2, j_9	j_5	j_6, j_7	Second Child
C_i	24	35	28	19	29	

Figure 6.1: Recombination process of the Random Grouping Crossover (RGX) operator

distribution. This rank strategy is the one used for the initial RGX that allows ranking the machines with an approach completely random. Similarly, Figure 6.2b includes the parent solutions with their machines ranked with the Average(p_i) strategy, which consists of calculating the average processing time required by each machine i to process the n jobs of the instance I, to later ranking the machines from smallest to largest based on such average. In this way, each pair of parents selected to generate offspring always gives preference to the fastest machine, that is, the machine that on average could process the n jobs faster by itself. Likewise, it saves the slowest machine for last. Therefore, the machines in all the parents are always arranged in the same way before performing the transmission process. On the other hand, this study considers two rank strategies based on one criterion related to the current status of the machines, called N_{jobs} and C_i. Figure 6.2c contains the parent solutions with their machines ranked according to the number of jobs assigned N_{jobs} to each machine, from highest to lowest. It is important to note that, if two or more machines have the same number of jobs, they are ranked in non-decreasing order according to their index, i.e., i_1, i_2, ..., i_m. Similarly, 6.2d includes the parent solutions with their machines ranked with a strategy C_i, that ranks the machines of parents in non-decreasing order based on their C_i values, giving preference to machines with less processing time. Like in the strategy N_{jobs}, if two or more machines have the same processing time p_{ij}, they are ranked in non-decreasing order according to their index i.

Finally, we study the performance of two strategies that arrange the machines based on two criteria: N_{jobs}-C_i and C_i-N_{jobs}. Figure 6.2e shows the machines in the parent solutions ranked according to the number of jobs. Unlike the N_{jobs} strategy, in this case, the machines with the same number of jobs are sorted based on the second criterion, the processing time of the machines C_i. In this way, this strategy first gives preference to the machines with the highest number of jobs and selects the machine with the lowest C_i among two machines with the same number of jobs. In contrast, the strategy C_i-N_{jobs}, first arranges the machines based on the processing time (from lowest to highest C_i value), and if there is a tie, it uses the number of jobs N_{jobs} of the machines as tiebreaker, placing first the machine with the highest number of jobs. Figure 6.2f includes the parent solutions with their machines ranked with this strategy.

Once the six strategies to order the groups were designed, we incorporated them into the initial RGX operator, giving rise to six crossover operators identified with their arrangement strategy: Permutation, Average(p_i), N_{jobs}, C_i, N_{jobs}-C_i, and C_i-N_{jobs}. Thus, each one of these variants starts its recombination process by arranging the machines of the parents in question, based on one of the six studied strategies. Subsequently, they proceed with the gene transmission process of RGX, described above. The suitability of the six proposed strategies was evaluated employing the proposed experimental design. That is, generating 100 solutions with the Min() heuristic, using the same seed, recombining the genetic material of the 100 solutions in each generation, utilizing elitist replacement, and employing 500 generations. Likewise, we compare the performance of the six variants based on their RPD to CPLEX. Table 6.5 shows the experimental results of this phase. The first two columns indicate the criteria used to group the instances, i.e., n, m, p_{ij}, and the 1400 instances together. Thus, the remaining columns contain the average RPD obtained by each crossover

Given two parent solutions, and a test instance I:

Machines	i_1	i_2	i_3	i_4	i_5	
Jobs	j_7, j_9	j_4	j_5, j_6, j_8	j_2	j_1, j_3, j_{10}	First Parent
C_i	29	10	20	18	7	

Machines	i_1	i_2	i_3	i_4	i_5	
Jobs	j_2, j_8, j_{10}	j_4, j_5	j_1, j_3	j_6, j_9	j_7	Second Parent
C_i	30	30	12	23	23	

Test Instance I

m \ n	i_1	i_2	i_3	i_4	i_5
j_1	16	15	6	25	5
j_2	10	10	16	18	8
j_3	18	15	6	10	1
j_4	15	10	16	20	12
j_5	29	20	5	19	29
j_6	20	12	5	16	6
j_7	11	19	3	3	23
j_8	12	10	10	20	10
j_9	28	20	12	7	27
j_{10}	8	20	18	28	2

The six strategies to rank the machines in the parent solutions work as follows:

Permutation

First Parent

Machines	i_2	i_4	i_1	i_3	i_5
Jobs	j_4	j_2	j_7, j_9	j_5, j_6, j_8	j_1, j_3, j_{10}
C_i	10	18	29	20	7

Second Parent

Machines	i_5	i_3	i_4	i_1	i_2
Jobs	j_7	j_1, j_3	j_6, j_9	j_2, j_8, j_{10}	j_4, j_5
C_i	23	12	23	30	30

Average(p_i)

Average machines	i_1	i_2	i_3	i_4	i_5
	16.7	15.1	9.7	16.6	12.3

Sorted machines	i_3	i_5	i_2	i_4	i_1
	9.7	12.3	15.1	16.6	16.7

First Parent

Machines	i_3	i_5	i_2	i_4	i_1
Jobs	j_5, j_6, j_8	j_1, j_3, j_{10}	j_4	j_2	j_7, j_9
C_i	20	7	10	18	29

Second Parent

Machines	i_3	i_5	i_2	i_4	i_1
Jobs	j_1, j_3	j_7	j_4, j_5	j_6, j_9	j_2, j_8, j_{10}
C_i	12	23	30	23	30

N_{jobs}

First Parent

Machines	i_3	i_5	i_1	i_2	i_4
Jobs	j_5, j_6, j_8	j_1, j_3, j_{10}	j_7, j_9	j_4	j_2
C_i	20	7	29	10	18

Second Parent

Machines	i_1	i_3	i_4	i_2	i_5
Jobs	j_2, j_8, j_{10}	j_1, j_3	j_6, j_9	j_4, j_5	j_7
C_i	30	12	23	30	23

C_i

First Parent

Machines	i_5	i_2	i_4	i_3	i_1
Jobs	j_1, j_3, j_{10}	j_4	j_2	j_5, j_6, j_8	j_7, j_9
C_i	7	10	18	20	29

Second Parent

Machines	i_3	i_4	i_5	i_2	i_1
Jobs	j_1, j_3	j_6, j_9	j_7	j_4, j_5	j_2, j_8, j_{10}
C_i	12	23	23	30	30

N_{jobs}-C_i

First Parent

Machines	i_5	i_3	i_1	i_2	i_4
Jobs	j_1, j_3, j_{10}	j_5, j_6, j_8	j_7, j_9	j_4	j_2
C_i	7	20	29	10	18

Second Parent

Machines	i_1	i_3	i_4	i_2	i_5
Jobs	j_2, j_8, j_{10}	j_1, j_3	j_6, j_9	j_4, j_5	j_7
C_i	30	12	23	30	23

C_i-N_{jobs}

First Parent

Machines	i_5	i_2	i_4	i_3	i_1
Jobs	j_1, j_3, j_{10}	j_4	j_2	j_5, j_6, j_8	j_7, j_9
C_i	7	10	18	20	29

Second Parent

Machines	i_3	i_4	i_5	i_1	i_2
Jobs	j_1, j_3	j_6, j_9	j_7	j_2, j_8, j_{10}	j_4, j_5
C_i	12	23	23	30	30

Figure 6.2: Strategies to rank the machines in parent solutions

operator for each grouping criterion, highlighting in bold the best results.

As Table 6.5 indicates, the best variants are C_i and C_i-N_{jobs}, obtaining better results even for all the grouping criteria. It is important to note that given this behavior, we studied the optimization process of the two variants C_i and C_i-N_{jobs} in the following phases of the experimental study. However, to give a better structure to the next sections of the document, only the results of the variant C_i-N_{jobs} will be presented

Table 6.5: Comparison of the crossover operators: Permutation, Average(p_i), N_{jobs}, C_i, N_{jobs}-C_i, and C_i-N_{jobs} using RPD.

Instance Set		**Permutation**	**Average(P_i)**	C_i	N_{jobs}	C_i-N_{jobs}	N_{jobs}-C_i
n	100	0.191	0.259	0.166	0.333	**0.165**	0.192
	200	0.216	0.258	**0.191**	0.308	0.192	0.210
	500	0.192	0.208	**0.182**	0.248	**0.182**	0.200
	1000	0.183	0.188	**0.178**	0.226	**0.178**	0.201
m	10	0.190	0.200	**0.183**	0.261	**0.183**	0.212
	20	0.201	0.233	0.184	0.290	**0.183**	0.212
	30	0.186	0.223	**0.168**	0.263	**0.168**	0.185
	40	0.191	0.231	**0.173**	0.278	**0.173**	0.184
	50	0.210	0.254	**0.188**	0.300	0.189	0.211
p_{ij}	U(1, 100)	0.456	0.591	0.394	0.701	**0.392**	0.455
	U(10, 100)	0.323	0.390	0.290	0.440	0.291	0.325
	U(100, 120)	0.053	0.055	**0.051**	0.081	**0.051**	0.057
	U(100, 200)	0.166	0.176	**0.161**	0.212	**0.161**	0.170
	U(1000, 1100)	0.027	0.028	**0.026**	0.043	**0.026**	0.029
	$JobCorre$	0.155	0.151	0.158	0.224	**0.157**	0.159
	$MacCorre$	0.188	0.208	**0.175**	0.250	0.176	0.212
1400 instances		0.196	0.228	**0.179**	0.279	**0.179**	0.201

since it showed better results in the subsequent phases.

6.2.3 Strategies to establish the machine transmission order and the number of children

After identifying the best option to sort the machines of the parent solutions, we explore different ways of conducting the genetic material transmission from parents to offspring and the number of children to generate from a pair of parents. In this way, we present twelve new crossover operators. All of them start with the strategy to rank the machines C_i - N_{jobs}, but they differ in the scheme used to perform the transmission of the genes. Six operators generate a child from two parents, referred to as One machine Min(C_i), Two machines Min(C_i), One machine Max(N_{jobs}), Two machines Max(N_{jobs}), One machine Random(), and Two machines Random(). Additionally, we introduce six operators that generate two children from two parents, named Max(N_{jobs}) Fixed, Max(N_{jobs}) Random, Min(C_i) Fixed, Min(C_i) Random(), and Random Switch().

Figure 6.3 shows two potential parent solutions to an instance I with five machines from i_1 to i_5 and ten jobs from j_1 to j_{10}. Each parent contains the ten jobs distributed among the five machines and a vector C_i to keep track of the processing time allocated to each machine i. Additionally, the figure shows the machines of the two parents ranked according to the strategy C_i-N_{jobs}, which first arranges the machines in non-decreasing order according to their processing time C_i and then rearranges the machines tied in time according to the number of jobs assigned to them, giving priority to those with the highest number of jobs. Finally, this figure includes a set of arrows to indicate that the arranged machines will be compared in parallel. That is, the first machine i_5 of the first parent against the first machine i_3 of the second parent, the second machine i_2 of the first parent against the second machine i_5 from the second parent, and so on. In this way, the transmission order of each pair of machines will be determined by the decision criterion used by each of the twelve strategies studied.

Two parent solutions with their vectors arranged using the Strategy C_i-N_{jobs}, and the instance I that they solve.

Machines	i_5	i_2	i_4	i_3	i_1	First Parent
Jobs	j_1, j_3, j_{10}	j_4	j_2	j_5, j_6, j_8	j_7, j_9	
C_i	7	10	18	20	29	
Machines	i_3	i_5	i_4	i_2	i_1	Second Parent
Jobs	j_1, j_3, j_7	j_2, j_{10}	j_6	j_5	j_4, j_8, j_9	
C_i	15	10	16	20	55	

Test Instance I

m \ n	i_1	i_2	i_3	i_4	i_5
j_1	16	15	6	25	5
j_2	10	10	16	18	8
j_3	18	15	6	10	1
j_4	15	10	16	20	12
j_5	29	20	5	19	29
j_6	20	12	5	16	6
j_7	11	19	3	3	23
j_8	12	10	10	20	10
j_9	28	20	12	7	27
j_{10}	8	20	18	28	2

Figure 6.3: Two parent solutions to explain the genetic material transmission process of the twelve proposed strategies.

Figure 6.4 shows the procedures of the two strategies that two parents use to generate a

child based on the Random() criterion. Figure 6.4a contains the result of transmitting the machines from the two parents P_1 and P_2 (described in Figure 6.3) to a child C with the gene transmission strategy One Machine Random(). For each pair of machines, this strategy uses a random probability p generated with a uniform distribution to determine which of the two parent machines should be transmitted to the child. Figure 6.4a shows an example that includes the two parents with their machines ranked using the strategy C_i-N_{jobs}, a set of arrows indicating the pairs of machines compared in parallel, and a random probability value p for each of those pairs. Given this information, the strategy One Machine Random() builds the child C as follows. The p value of each machine pair determines whether the child receives the machine either from the first parent P_1 or the one from the second parent P_2. If $p \leq 0.5$ the child receives the machine from the first parent P_1, otherwise, it receives the machine from the second parent P_2. During the transmission process, the child receives the machines according to the random probabilities, discarding the repeated machines that appear in second place and the ones with at least a repeated job (highlighted in red). Usually, the result of this process is an infeasible child without some jobs (placed in the MJ box) and machines missed during the transmission process. Thus, it is necessary to use the Min() assignment heuristic to transform the child into a feasible one by re-inserting the missed jobs.

Like One Machine Random(), the Two Machines Random() strategy uses a random probability p generated with a uniform distribution to establish the transmission order of each machine pair. However, this strategy transmits the machines of both parents p_1 and p_2 to the child C. Figure 6.4b shows an example that includes the two parents with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and a probability p for each of those pairs. Thus, the strategy Two Machines Random() procedure to generate a child is as follows. The random probabilities determine which machine of each pair is transmitted first. If $p \leq 0.5$ the child C receives the machine from the first parent P_1 and then the one from the second parent P_2. Otherwise, the transmission order of the machines is reversed. In this sense, the child C receives the machines in the order generated by the random probabilities, deleting the repeated machines that appear in second place and the ones with at least a repeated job (highlighted in red). Frequently, this process generates an infeasible child without some jobs (placed in the MJ box) and machines missed during the recombination process. Thus, the missed jobs are re-inserted with the assignment heuristic Min().

Additionally, we studied two machine transmission strategies that use two parents P_1 and P_2 to generate a child C according to the Max(N_{jobs}) criterion, called One Machine Max(N_{jobs}) and Two Machine Max(N_{jobs}). For each pair of machines, these strategies give priority to the machine with the highest number of jobs. Figure 6.5 contains the detail of the procedures of these strategies. Figure 6.5a shows the resulting child of transmitting the machines from the two parents (described in Figure 6.3) to a child with the One Machine Max(N_{jobs}) gene transmission strategy. The example contains two parents P_1 and P_2 with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability p values for pairs of machines with the same number of jobs. Given the above, the One Machine Max(N_{jobs}) strategy

Given the two parent solutions and the instance I of Figure 6.3, the gene transmission process of the strategies that generate a child from two parents with the Random() criterion works as follows.

a) One Machine Random()

Machines	i_5	i_2	i_4	i_3	i_1	
Jobs	j_1, j_3, j_{10}	j_4	j_2	j_5, j_6, j_8	j_7, j_9	First Parent
C_i	7	10	18	20	29	

p=0.1 p=0.4 p=0.9 p=0.7 p=0.2

Machines	i_3	i_5	i_4	i_2	i_1	
Jobs	j_1, j_3, j_7	j_2, j_{10}	j_6	j_5	j_4, j_8, j_9	Second Parent
C_i	15	10	16	20	55	

Machines	i_5	i_2	i_4	i_1		
Jobs	j_1, j_3, j_{10}	j_4	j_6	j_7, j_9	Child	MJ: j_2, j_5, j_8
C_i	7	10	16	29		

b) Two Machines Random()

Machines	i_5	i_2	i_4	i_3	i_1	
Jobs	j_1, j_3, j_{10}	j_4	j_2	j_5, j_6, j_8	j_7, j_9	First Parent
C_i	7	10	18	20	29	

p=0.1 p=0.4 p=0.9 p=0.7 p=0.2

Machines	i_3	i_5	i_4	i_2	i_1	
Jobs	j_1, j_3, j_7	j_2, j_{10}	j_6	j_5	j_4, j_8, j_9	Second Parent
C_i	15	10	16	20	55	

Machines	i_5	i_2	i_4	i_1		
Jobs	j_1, j_3, j_{10}	j_4	j_6	j_7, j_9	Child	MJ: j_2, j_5, j_8
C_i	7	10	16	29		

Figure 6.4: Machine transmission strategies that use two parents to generate a child based on the Random() criterion.

works as follows. For each pair of machines, this strategy only transmits the machine with the highest number of jobs (outlined in blue) to the child C. If both machines have the same number of jobs (the two machines are outlined in blue), this strategy generates a random probability value p with a uniform distribution to determine which of the machines is transmitted to the child C. If $p \leq 0.5$ the child receives the machine from the first parent P_1, otherwise, it receives the machine from the second parent P_2. Furthermore, before transmitting each machine, One Machine Max(N_{jobs}) verifies that it has not been transmitted by the other parent yet. Otherwise, it is discarded. Likewise, machines with one or more repeated jobs (highlighted in red) are discarded. Usually, the resulted child is infeasible since some jobs, placed in the box MJ, can be missed during the transmission process. Hence, it is necessary to use the Min() assignment heuristic to re-insert them into the child solution.

Similar to One Machine Max(N_{jobs}), the Two Machines Max(N_{jobs}) strategy gives priority to the machine with the highest number of jobs. However, this strategy transmits the machines of both parents P_1 and P_2 to the child C. Figure 6.5b presents an example that consists of two parents P_1 and P_2 with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability

values p for the pairs of machines tied with the same number of jobs. Thus, the Two Machines Max(N_{jobs}) strategy to generate a child C works as follows. For each pair of machines, this strategy first transmits the machine with the highest number of jobs (outlined in blue) to the child C, and later the other one. If both machines have the same number of jobs (the two machines are outlined in blue), this strategy generates a random probability value p with a uniform distribution to determine the transmission order. If $p \leq 0.5$, the child C receives first the machine from the first parent P_1 and later the one of the second parent P_2; otherwise, the transmission order of the machines is reversed. Moreover, in order to avoid repeated genetic material, Two Machines Max(N_{jobs}) checks that the machine to transmit has not been inherited by the other parent yet. Otherwise, it is discarded. Alike, it discards the machines with one or more repeated jobs (highlighted in red). Usually, the generated child is infeasible since some jobs, placed in the box MJ, can be missed during the transmission process. Hence, it is necessary to use the Min() assignment heuristic to re-insert them into the solution.

Given the two parent solutions and the instance I of Figure 6.3, the gene transmission process of the strategies that generate a child from two parents with the Max(N_{jobs}) criterion works as follows.

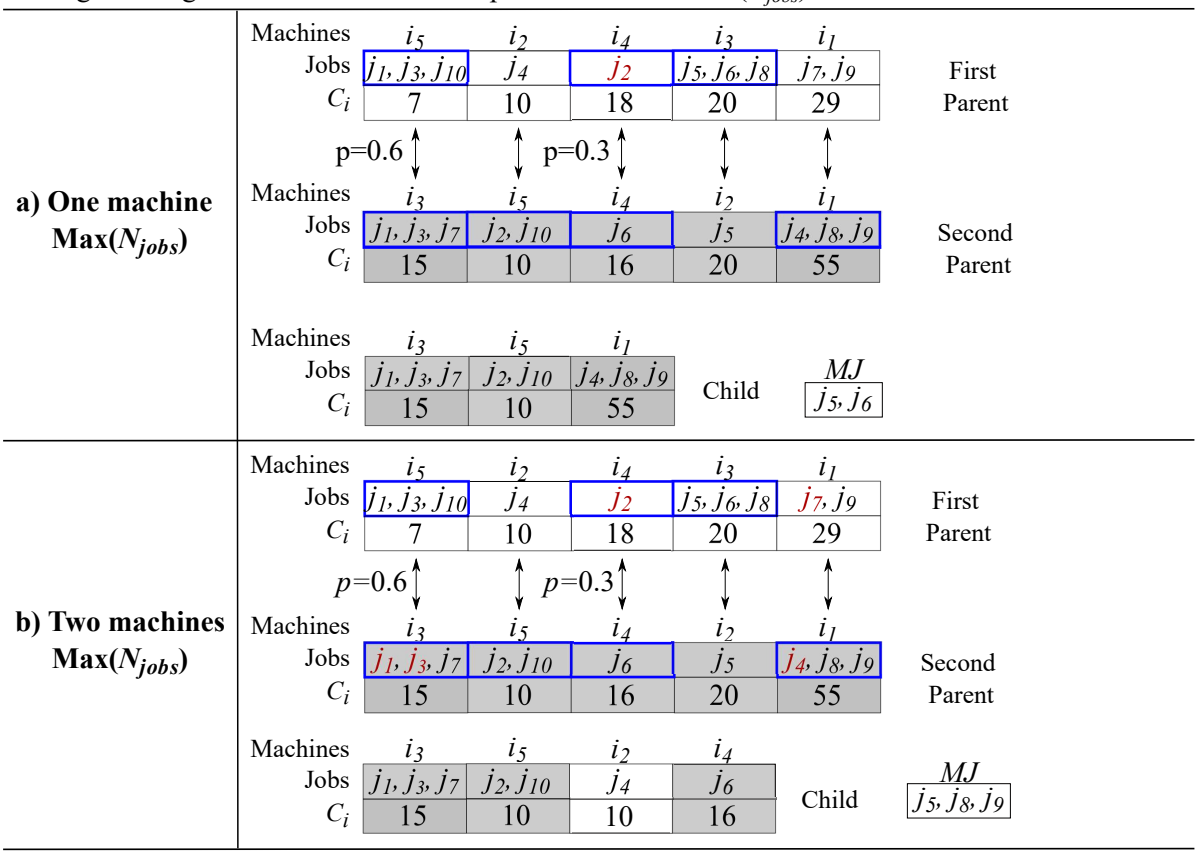


Figure 6.5: Machine transmission strategies that use two parents to generate a child based on the Max(N_{jobs}) criterion.

Finally, we studied two machine transmission strategies that use two parents P_1 and P_2 to generate a child C according to the Min(C_i) criterion, referred to as One Machine Min(C_i) and Two Machine Min(C_i). For each pair of machines, these strategies give

priority to the machine that requires the shortest time to process its jobs. Figure 6.6 contains the detail of the procedures of these strategies. Figure 6.6a shows the resulting child C of transmitting the machines from the two parents $P_1$and P_2 (described in Figure 6.3) to a child C with the One Machine Min(C_i) gene transmission strategy. The example contains two parents P_1 and P_2 with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability p values for the pairs of machines with the processing time. Given the above, the One Machine Min(C_i) strategy works as follows. For each pair of machines, this strategy only transmits the machine with the shortest processing time (outlined in blue) to the child C. If both machines have the same processing time (the two machines are outlined in blue), this strategy generates a random probability value p with a uniform distribution to determine which of the machines is transmitted to the child C. If $p \leq 0.5$ the child C receives the machine from the first parent P_1, otherwise, it receives the machine from the second parent P_2. Furthermore, before transmitting each machine, One Machine Min(C_i) verifies that it has not been transmitted by the other parent yet. Otherwise, it is discarded. Likewise, machines with one or more repeated jobs (highlighted in red) are discarded. Usually, the resulted child is infeasible since some jobs, placed in the box MJ, can be missed during the transmission process. Hence, it is necessary to use the assignment heuristic Min() to re-insert them into the solution.

Like One Machine Min(C_i), the Two Machines Min(C_i) strategy gives priority to the machine with the shortest processing time. However, this strategy transmits the machines of both parents P_1 and P_2 to the child C. Figure 6.6b presents an example that consist of two parents with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability values p for the pairs of machines tied with the same number of jobs. Thus, the Two Machines Min(C_i) strategy generates a child C as follows. For each pair of machines, this strategy first transmits the machine with the shortest processing time (outlined in blue) to the child C, and later the other one. If both machines have the same number of jobs (the two machines are outlined in blue), this strategy generates a random probability value p with a uniform distribution to determine the transmission order. If $p \leq 0.5$ the child C receives first the machine from the first parent P_1 and later the one of the second parent P_2; otherwise, the transmission order of the machines is reversed. Moreover, in order to avoid repeated genetic material, Two Machines Min(C_i) checks that the machine to transmit has not been inherited by the other parent yet. Otherwise, it is discarded. Alike, it discards the machines with one or more repeated jobs (highlighted in red). Usually, the generated child is infeasible since some jobs, placed in the box MJ, can be missed during the transmission process. Hence, it is necessary to use the Min() assignment heuristic to re-insert them into the solution.

On the other hand, we also designed six machine transmission strategies that use two parents P_1 and P_2 to generate two children C_1 and C_2. Figure 6.7 shows the procedures of the first two strategies that use the Random() criterion to create two children C_1 and C_2 from two parents P_1 and P_2, called Random() and Random Switch(). Figure 6.7a contains the result of transmitting the machines from the two parents to each child with the Random() gene transmission strategy. For each pair of machines, this strategy uses a random probability p generated with a uniform distribution to determine which

Given the two parent solutions and the instance I of Figure 6.3, the gene transmission process of the strategies that generate a child from two parents with the Min(C_i) criterion works as follows.

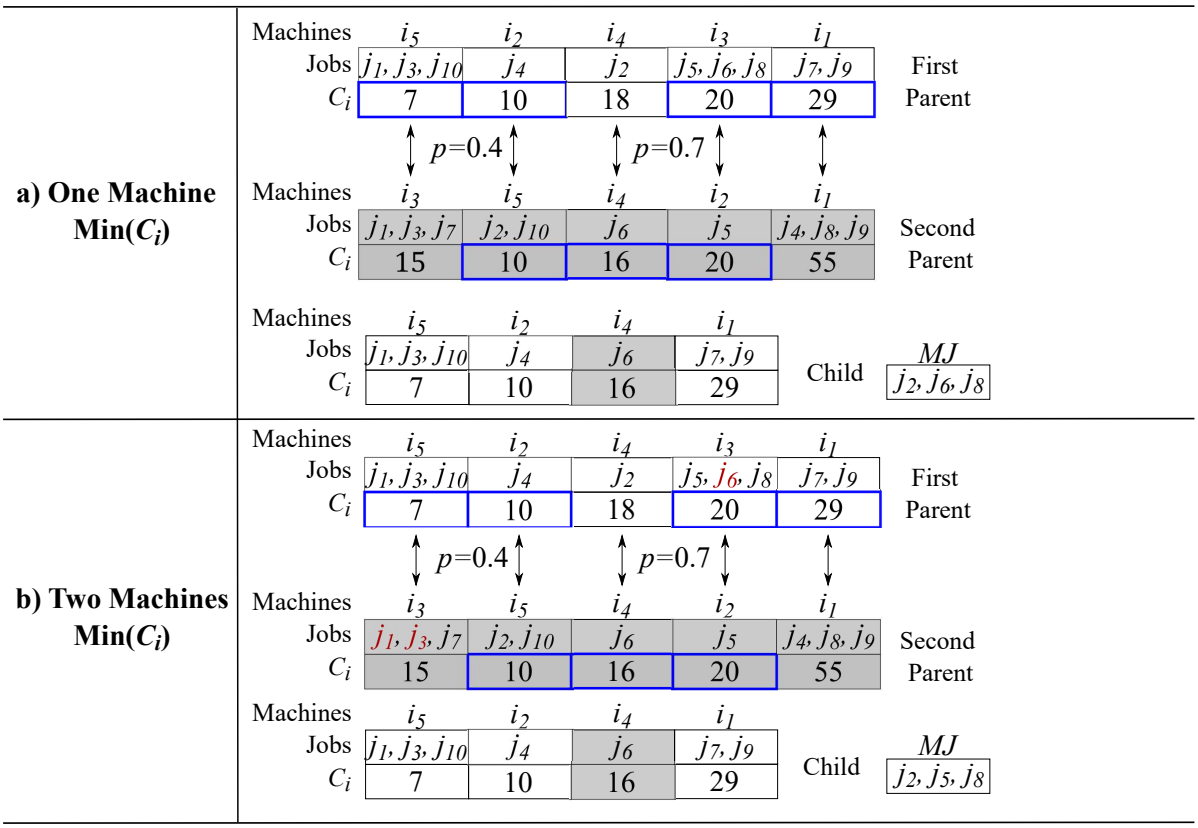


Figure 6.6: Machine transmission strategies that use two parents to generate a child based on the Min(C_i) criterion.

of the two machines should be transmitted first to each child. Figure 6.7a shows an example that includes the two parents ranked with the strategy C_i-N_{jobs}, a set of arrows indicating the pairs of machines compared in parallel, and a random probability value p for each of those pairs. Given this information, the Random() strategy builds the children as follows. The p value of each machine pair determines the transmission order of the machines in the following way. If $p \leq 0.5$ the first child C_1 first receives the machine from the first parent P_1 and later the machine of the second parent P_2, while the second child C_2 receives first the machine from the second parent P_2 and later the one from the first parent P_1. Otherwise, the first child C_1 first receives the machine from the second parent P_2 and later the machine from the first parent P_1, while the second child C_2 first receives the machine from the first parent P_1 and later the one from the second parent P_2. During the transmission process, the children receive the machines according to the random probabilities, discarding the repeated machines that appear in second place and the ones with at least a repeated job (highlighted in red). Usually, the results of this process are two infeasible children without some jobs (placed in the box of misses jobs MJ) and machines missed during the transmission process. Thus, it is necessary to use the Min() assignment heuristic to transform the children

into feasible ones by re-inserting the missed jobs.

Similar to Random(), the Random Switch() strategy uses a random probability p generated with a uniform distribution to establishes the transmission order of each machine pair. However, this strategy only transmits one of the machines to the children. Figure 6.7b shows an example that includes the two parents P_1 and P_2 with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and a probability p for each of those pairs. Thus, the strategy Random Switch() procedure to generate a child is as follows. The random probabilities determine which machine of each pair is transmitted to each child. If $p \leq 0.5$ the first child C_1 receives the machine from the first parent C_1 and the second child C_2 receives the machine from the second parent P_2. Otherwise, the transmission order of the machines is reversed, i.e., the first child C_1 receives the machine from the second parent P_2 and the second child C_2 receives the machine from the first parent P_1. Therefore, the children receive the machines in the order established by the random probabilities, discarding the repeated machines that appear in second place and the ones with at least a repeated job (highlighted in red). Frequently, this process generates infeasible children without some jobs (placed in the box of missed jobs MJ) and machines missed during the recombination process. Thus, the missed jobs are re-inserted with the assignment heuristic Min().

Additionally, we designed two machine transmission strategies that use two parents to generate two children according to the Max(N_{jobs}) criterion, called Max(N_{jobs}) Fixed and Max(N_{jobs}) Random. For each pair of machines, these strategies give priority to the machine with the highest number of jobs. Figure 6.8 contains the detail of the procedures of these strategies. Figure 6.8a shows the resulting children of transmitting the machines from the two parents with the Max(N_{jobs}) Fixed machine transmission strategy. The example contains two parents P_1 and P_2 with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability p values for pairs of machines with the same number of jobs. Given the above, the Max(N_{jobs}) Fixed strategy works as follows. For each pair of machines, this strategy first transmits the machine with the highest number of jobs (outlined in blue) to both children C_1 and C_2, and later the other one. If both machines have the same number of jobs (the two machines are outlined in blue), the first child C_1 first receives the machine from the first parent P_1 and later the machine of the second parent P_2, while the second child C_2 first receives the machine from the second parent P_2 and later the one from the first parent P_1. Furthermore, before transmitting each machine, Max(N_{jobs}) Fixed verifies that it has not been transmitted by the other parent yet. Otherwise, it is discarded. Likewise, machines with one or more repeated jobs (highlighted in red) are discarded. Usually, the resulted children are infeasible since some jobs can be missed during the transmission process (placed in the box of missed jobs MJ). Hence, it is necessary to use the Min() assignment heuristic to re-insert them into the solution.

Like Max(N_{jobs}) Fixed, the Max(N_{jobs}) Random strategy gives priority to the machine with the highest number of jobs. However, this strategy incorporates randomness to establishes the order of transmission of the machine pairs with the same number of jobs. Figure 6.8b presents an example that consist of two parents P_1 and P_2 with

Given the two parent solutions and the instance I of Figure 6.3, the gene transmission process of the strategies that generate two children from two parents with the Random() criterion works as follows.

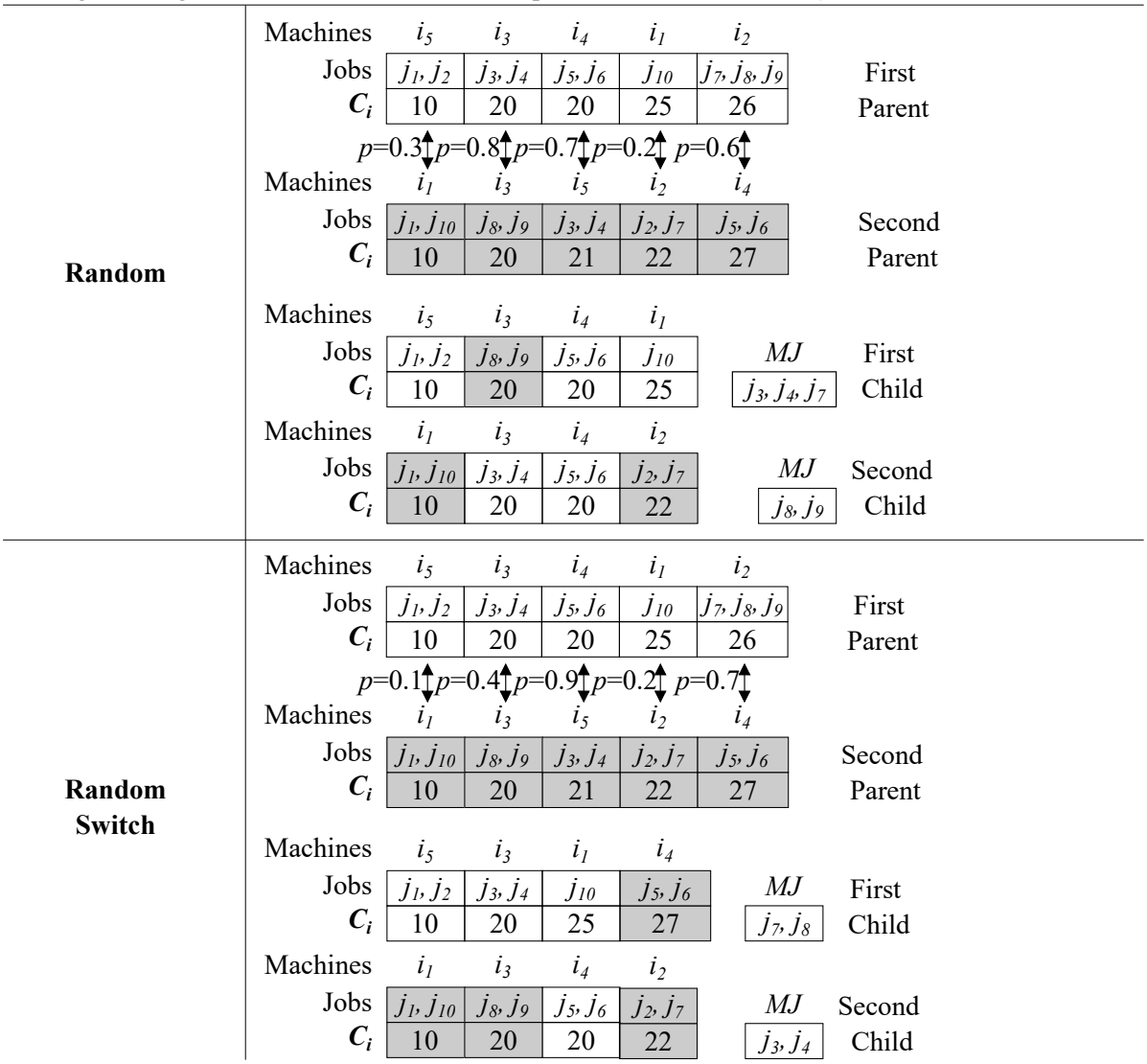


Figure 6.7: Machine transmission strategies that use two parents to generate two children based on the Random() criterion.

their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability values p for the pairs of machines with the same number of jobs. Thus, the Max(N_{jobs}) Fixed strategy procedure to generate a child is as follows. For each pair of machines, this strategy first transmits the machine with the highest number of jobs (outlined in blue) to both children C_1 and C_2, and later the other one. If both machines have the same number of jobs (the two machines are outlined in blue), Max(N_{jobs}) Fixed generates a random probability p with a uniform distribution to establish the order of transmission of the machines. If $p \leq 0.5$ the first child C_1 first receives the machine from the first parent P_1 and later the machine of the second parent P_2, while the second child C_2 first receives the machine from the second

parent P_2 and later the one from the first parent P_1. Otherwise, the transmission order of the machines is reversed, i.e., the first child C_1 first receives the machine from the second parent P_2 and later the machine from the first parent P_1, while the second child C_2 first receives the machine from the first parent P_1 and later the one from the second parent P_2. Moreover, in order to avoid repeated genetic material, Max(N_{jobs}) Random checks that the machine to transmit has not been inherited by the other parent yet. Otherwise, it is discarded. Alike, it discards the machines with one or more repeated jobs (highlighted in red). Usually, the generated children are infeasible since some jobs (placed in the box of missed jobs MJ) can be missed during the transmission process. Hence, it is necessary to use the assignment heuristic Min() to re-insert them into the solution.

Finally, we designed two machine transmission strategies that use two parents to generate two children according to the Min(C_i) criterion, called Min(C_i) Fixed and Min(C_i) Random. For each pair of machines, these strategies give priority to the machine that requires the shortest time to process its jobs. Figure 6.9 contains the detail of the procedures of these strategies. Figure 6.9a shows the resulting children of transmitting the machines from the two parents (described in Figure 6.3) with the Min(C_i) Fixed transmission strategy. The example contains two parents with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability p values for the pairs of machines with the processing time. Given the above, the Min(C_i) Fixed strategy works as follows. For each pair of machines, this strategy first transmits the machine with the shortest processing time (outlined in blue) to both children, and later the other one. If both machines have the same processing time (the two machines are outlined in blue), the first child first receives the machine from the first parent and later the machine from the second parent, while the second child first receives the machine from the second parent and later the one from the first parent. Furthermore, before transmitting each machine, Min(C_i) Fixed verifies that it has not been transmitted by the other parent yet. Otherwise, it is discarded. Likewise, machines with one or more repeated jobs (highlighted in red) are discarded. Usually, the resulted children are infeasible since some jobs (placed in the box of missed jobs MJ) can be missed during the transmission process. Hence, it is necessary to use the assignment heuristic Min() to re-insert them into the solution.

Like Min(C_i) Fixed, the Min(C_i) Random strategy gives priority to the machine with the shortest processing time. However, this strategy incorporates randomness to establishes the order of transmission of the machine pairs with the same processing time. Figure 6.9b presents an example that consist of two parents with their machines ranked with the C_i-N_{jobs} strategy, a set of arrows indicating the pairs of machines, and some probability values p for the pairs of machines with the same processing time. Thus, the Min(C_i) Random strategy procedure to generate two children is as follows. For each pair of machines, this strategy first transmits the machine with the shortest processing time (outlined in blue) to both children, and later the other one. If both machines have the same processing time (the two machines are outlined in blue), Min(C_i) Random generates a random probability p with a uniform distribution to establish the order of transmission of the machines. If $p \leq 0.5$ the first child first receives the machine from the first parent and later the machine of the second parent, while the second

Given the two parent solutions and the instance I of Figure 6.3, the gene transmission process of the strategies that generate two children from two parents with the Max(N_{jobs}) criterion works as follows.

Max(N_{jobs}) Fixed

Machines	i_5	i_3	i_4	i_1	i_2	
Jobs	j_1, j_2	j_3, j_4	j_5, j_6	j_{10}	j_7, j_8, j_9	First Parent
C_i	10	20	20	25	26	
Machines	i_1	i_3	i_5	i_2	i_4	
Jobs	j_1, j_{10}	j_8, j_9	j_3, j_4	j_2, j_7	j_5, j_6	Second Parent
C_i	10	20	21	22	27	

Machines	i_5	i_3	i_4	i_1		
Jobs	j_1, j_2	j_3, j_4	j_5, j_6	j_{10}	MJ	First Child
C_i	10	20	20	25	j_7, j_8, j_9	
Machines	i_1	i_3	i_5	i_4	i_2	
Jobs	j_1, j_{10}	j_8, j_9	j_3, j_4	j_5, j_6	j_2, j_7	Second Child
C_i	10	20	21	20	22	

Max(N_{jobs}) Random

Machines	i_5	i_3	i_4	i_1	i_2	
Jobs	j_1, j_2	j_3, j_4	j_5, j_6	j_{10}	j_7, j_8, j_9	First Parent
C_i	10	20	20	25	26	
Machines	i_1	i_3	i_5	i_2	i_4	
Jobs	j_1, j_{10}	j_8, j_9	j_3, j_4	j_2, j_7	j_5, j_6	Second Parent
C_i	10	20	21	22	27	

Machines	i_5	i_3	i_4	i_1		
Jobs	j_1, j_2	j_8, j_9	j_5, j_6	j_{10}	MJ	First Child
C_i	10	20	20	25	j_3, j_4, j_7	
Machines	i_1	i_3	i_4	i_2		
Jobs	j_1, j_{10}	j_3, j_4	j_5, j_6	j_2, j_7	MJ	Second Child
C_i	10	20	20	22	j_8, j_9	

Figure 6.8: Machine transmission strategies that use two parents to generate two children based on the Max(N_{jobs}) criterion.

child first receives the machine from the second parent and later the one from the first parent. Otherwise, the transmission order of the machines is reversed, i.e., the first child first receives the machine from the second parent and later the machine from the first parent, while the second child first receives the machine from the first parent and later the one from the second parent. Moreover, in order to avoid repeated genetic material, Min(C_i) Random checks that the machine to transmit has not been inherited by the other parent yet. Otherwise, it is discarded. Alike, it discards the machines with one or more repeated jobs (highlighted in red). Usually, the generated children are infeasible since some jobs (placed in the box of missed jobs MJ) can be missed

Table 6.6: Performance comparison of the machine transmission strategies: One machine Random(), Two machines Random(), One machine Max(N_{jobs}), Two machines Max(N_{jobs}), One machine Min(C_i), Two machines Min(C_i) based on the average RPD.

Instance Set		One machine Random	Two machines Random	One machine Max(n_{jobs})	Two machines Max(n_{jobs})	One machine Min(C_i)	Two machines Min(C_i)
n	100	0.170	0.210	0.159	0.213	**0.121**	0.201
	200	0.168	0.203	0.156	0.205	**0.121**	0.195
	500	0.157	0.191	0.148	0.193	**0.119**	0.184
	1000	0.181	0.215	0.168	0.221	**0.129**	0.206
m	10	0.180	0.221	0.169	0.225	**0.128**	0.212
	20	0.178	0.220	0.167	0.224	**0.128**	0.212
	30	0.176	0.218	0.166	0.221	**0.127**	0.210
	40	0.174	0.217	0.164	0.220	**0.126**	0.209
	50	0.173	0.215	0.162	0.218	**0.125**	0.207
p_{ij}	U(1, 100)	0.396	0.490	0.369	0.522	**0.224**	0.465
	U(10, 100)	0.291	0.324	0.250	0.321	**0.184**	0.296
	U(100, 120)	0.051	0.072	0.050	0.073	**0.047**	0.072
	U(100, 200)	0.161	0.189	0.150	0.189	**0.132**	0.185
	U(1000, 1100)	**0.026**	0.039	**0.026**	0.039	0.027	0.038
	$JobsCorre$	0.159	0.183	0.162	0.187	**0.147**	0.180
	$MacsCorre$	0.176	0.253	0.182	0.251	**0.143**	0.253
1400 instances		0.180	0.221	0.170	0.226	**0.129**	0.213

during the transmission process. Hence, it is necessary to use the Min() assignment heuristic to re-insert them into the solution.

Once the twelve strategies to establish the transmission order of the machine were designed, we incorporate them into the crossover operator under design, giving rise to twelve crossover operators. Each operator is identified by the transmission strategy that they use as One Machine Random(), Two Machines Random(), One Machine Max(N_{jobs}), Two Machines Max(N_{jobs}), Machine Min(C_i), Two Machines Min(C_i), Random(), Random Switch(), Max(N_{jobs}) Fixed, Max(N_{jobs}) Random, Min(C_i) Fixed, and Min(C_i) Random. Thus, each operator starts its recombination process by arranging the machines of the parents in question, based on arrangement strategy C_i-N_{jobs}. Subsequently, they proceed with one of the machine transmission process strategies. Like in the previous two phases, the suitability of the strategies studied was evaluated by generating 100 solutions with the heuristic Min(), using the same seed, recombining the genetic material of the 100 solutions in each generation, utilizing elitist replacement, and employing 500 generations. In the same way, we compare the performance of the twelve operators based on their RPD to CPLEX. Tables 6.6 and 6.7 show the experimental results of this phase. The first two columns indicate the criteria used to group the instances, i.e., n, m, p_{ij}, and the 1400 instances together. Thus, the remaining columns contain the average RPD obtained by each crossover operator for each grouping criterion, highlighting in bold the best results.

Given the two parent solutions and the instance I of Figure 6.3, the gene transmission process of the strategies that generate two children from two parents with the Min(C_i) criterion works as follows.

Min(C_i) Fixed

Machines	i_5	i_3	i_4	i_1	i_2	
Jobs	j_1, j_2	j_3, j_4	j_5, j_6	j_{10}	j_7, j_8, j_9	First Parent
C_i	10	20	20	25	26	

Machines	i_1	i_3	i_5	i_2	i_4	
Jobs	j_1, j_{10}	j_8, j_9	j_3, j_4	j_2, j_7	j_5, j_6	Second Parent
C_i	10	20	21	22	27	

Machines	i_5	i_3	i_4		
Jobs	j_1, j_2	j_3, j_4	j_5, j_6	*MJ*	First Child
C_i	10	20	20	j_7, j_8, j_9, j_{10}	

Machines	i_1	i_3	i_4	i_2		
Jobs	j_1, j_{10}	j_8, j_9	j_5, j_6	j_2, j_7	*MJ*	Second Child
C_i	10	20	20	22	j_3, j_4	

Min(C_i) Random

Machines	i_5	i_3	i_4	i_1	i_2	
Jobs	j_1, j_2	j_3, j_4	j_5, j_6	j_{10}	j_7, j_8, j_9	First Parent
C_i	10	20	20	25	26	

p=0.3 p

Machines	i_1	i_3	i_5	i_2	i_4	
Jobs	j_1, j_{10}	j_8, j_9	j_3, j_4	j_2, j_7	j_5, j_6	Second Parent
C_i	10	20	21	22	27	

Machines	i_5	i_3	i_4	i_1		
Jobs	j_1, j_2	j_8, j_9	j_5, j_6	j_{10}	*MJ*	First Child
C_i	10	20	20	25	j_3, j_4, j_7	

Machines	i_1	i_3	i_4	i_2		
Jobs	j_1, j_{10}	j_3, j_4	j_5, j_6	j_2, j_7	*MJ*	Second Child
C_i	10	20	20	22	j_8, j_9	

Figure 6.9: Machine transmission strategies that use two parents to generate two children based on the Min(C_i) criterion.

Table 6.7: Performance comparison of the machine transmission strategies: Random(), Random Switch(), Max(N_{jobs}) Fixed, Max(N_{jobs}) Random, Min(C_i) Fixed, Min(C_i) Random based on the average RPD.

Instance Set		**Min C_i Fixed**	**Min C_i) Random**	**Max n_{jobs} Fixed**	**Max n_{jobs} Random**	**Random**	**Random Switch**
n	100	0.206	**0.170**	0.213	0.213	0.201	0.202
	200	0.200	**0.168**	0.205	0.206	0.197	0.196
	500	0.189	**0.157**	0.192	0.193	0.188	0.186
	1000	0.212	**0.181**	0.222	0.222	0.206	0.207
m	10	0.217	**0.180**	0.227	0.227	0.212	0.213
	20	0.216	**0.178**	0.226	0.226	0.212	0.212
	30	0.215	**0.176**	0.221	0.222	0.210	0.211
	40	0.213	**0.174**	0.220	0.220	0.209	0.210
	50	0.211	**0.172**	0.218	0.218	0.207	0.208
pij	U(1, 100)	0.475	**0.395**	0.530	0.527	0.448	0.465
	U(10, 100)	0.322	**0.292**	0.329	0.330	0.309	0.303
	U(100, 120)	0.070	**0.051**	0.071	0.071	0.073	0.072
	U(100, 200)	0.186	**0.160**	0.185	0.186	0.185	0.185
	U(1000, 1100)	0.033	**0.026**	0.034	0.034	0.038	0.038
	$JobsCorre$	0.184	**0.160**	0.191	0.191	0.185	0.181
	$MacsCorre$	0.251	**0.176**	0.252	0.253	0.252	0.251
1400 instances		0.217	**0.180**	0.227	0.227	0.213	0.214

From Tables 6.6 and 6.7 can be observed that the operators that only generate one child obtained better results than those that generate two children. Moreover, Tables 6.6 indicates that the six operators obtained similar results. Therefore, all of them will be considered in the next experimental phase.

6.2.4 Strategies to handle the repeated jobs and machines

The last aspect related to the crossover process to be studied is the way the repeated genetic material is handled. As in the last stage, the operators that only generate one child obtained better results than those that generate two children, in this stage, we analyze the performance of the operators: One machine Random(), Two machines Random(), One machine Max(N_{jobs}), Two machines Max(N_{jobs}), One machine Min(C_i), Two machines Min(C_i), by replacing their strategy to handle the repeated genetic material Group Elimination (GE) by the Item Elimination (IE) strategy in order to identify the best option.

Instead of deleting all jobs in a machine with at least one repeating job, the Item elimination strategy removes only the repeated job. We incorporate them into the crossover operator under design, giving rise to six crossover operators. Each operator is identified by the transmission strategy that they use and the technique to handle the repeated genetic material as IE-One machine Random(), IE-Two machines Random(), IE-One machine Max(N_{jobs}), IE-Two machines Max(N_{jobs}), IE-One machine Min(C_i), IE-Two machines Min(C_i). Thus, each operator starts its recombination process by arranging the machines of the parents in question, based on arrangement strategy C_i-N_{jobs}. Subsequently, they proceed with one of the machine transmission process strategies. Finally, the six operators use the Item Elimination strategy. As in the previous phases, the suitability of the strategies studied was evaluated by generating 100 solutions with the heuristic Random min(), using the same seed, recombining the genetic material of the 100 solutions in each generation, utilizing elitist replacement, and employing 500 generations. In the same way, we compare the performance of the six operators based on their RPD to CPLEX. Table 6.8 shows the experimental results of this phase.

The study revealed that for each criterion used to transmit the machines, i.e., Random, Max(n_{jobs}), and Min(C_i), the variants that transmit only one machine work better with the Group Elimination genetic material handling technique. On the contrary, the variants that transmit both machines get better results with the Item Elimination technique. To analyze this phenomenon, we generate a graph with the average RPD reached by the six operators that generate a child in their two variants, i.e., one with Group Elimination technique and the other one with Item Elimination technique. Figure 6.10 shows the graph obtained, where the $x-$axis represents each operator, while the $y-$axis indicates the average RPD obtained by each operator. Each pair of blue and orange bars indicates an operator, the blue bars depict the operators with the Item Elimination technique, and the orange bar the operator with the Group Elimination technique. This graph reiterates that the operators that transmit the two machines to the children and use the item elimination handling technique have better results, highlighting the results achieved by the operator IE-Two Machines Min(C_i). However,

Table 6.8: Performance comparison of the genetic material handling technique item elimination in the machine transmission strategies: IE-One machine Random(), IE-Two machines Random(), IE-One machine Max(N_{jobs}), IE-Two machines Max(N_{jobs}), IE-One machine Min(C_i), IE-Two machines Min(C_i) based on the average RPD.

Instance Set		**IE-One machine Random**	**IE-Two machines Random**	**IE-One machine Max(n_{jobs})**	**IE-Two machines Max(n_{jobs})**	**IE-One machine Min(C_i)**	**IE-Two machines Min(C_i)**
n	100	0.208	0.170	0.155	0.170	0.168	**0.114**
	200	0.207	0.167	0.154	0.165	0.165	**0.113**
	500	0.198	0.165	0.149	0.160	0.161	**0.112**
	1000	0.232	0.159	0.168	0.160	0.157	**0.122**
m	10	0.230	0.173	0.169	0.172	0.170	**0.120**
	20	0.226	0.175	0.166	0.174	0.171	**0.119**
	30	0.222	0.177	0.164	0.175	0.173	**0.118**
	40	0.217	0.179	0.162	0.178	0.176	**0.117**
	50	0.213	0.181	0.160	0.179	0.178	**0.116**
p_{ij}	U(1, 100)	0.550	0.245	0.359	**0.270**	0.247	0.208
	U(10, 100)	0.389	0.159	0.235	0.148	0.156	**0.175**
	U(100, 120)	0.059	0.078	0.055	0.077	0.076	**0.043**
	U(100, 200)	0.190	0.189	0.148	0.184	0.186	**0.121**
	U(1000, 1100)	0.029	0.041	0.029	0.041	0.040	**0.026**
	$JobsCorre$	0.174	0.219	0.173	0.217	0.215	**0.150**
	$MacsCorre$	0.216	0.312	0.191	0.295	0.297	**0.125**
1400 instances		0.230	0.178	0.170	0.176	0.174	**0.121**

to get a bigger picture, in the next section we will look at the performance of four operators, the top two with the item elimination strategy and the top two with the group elimination strategy.

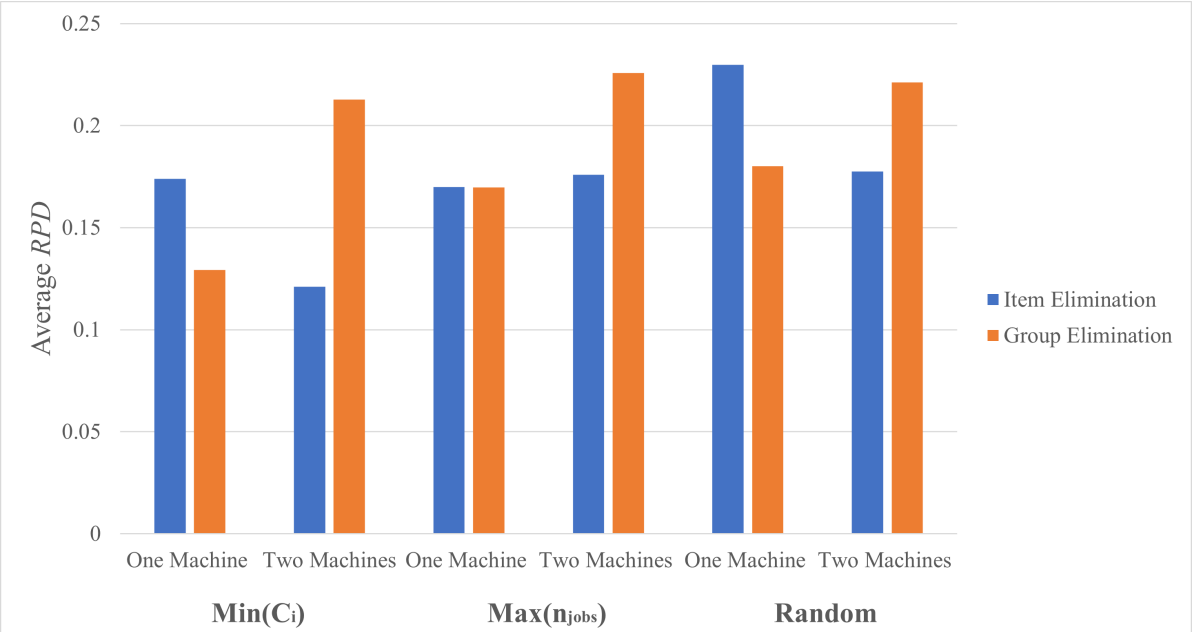


Figure 6.10: Performance comparison of the six operators that generate a child with the strategies to handle the genetic material Group Elimination and Item Elimination.

6.3 GGA with the old and the new crossover operators

In order to identify the most appropriate operator for the GGA presented in Chapter 4, at this stage we test the four variants that showed the best results in the two previous experimental studies, i.e., One machine Min(C_i) and Two machines Min (C_i) with the two techniques for handling constraints: Group Elimination (GE) and Item Elimination (IG). In this way, we analyze the performance of the GGA by replacing the old crossover operator with each of the four aforementioned operators, given rise to four new Enhanced GGAs (EGGAs). As all variants use the Min(C_i) criterion to transmit the machines, each EGGA variant is identified considering only the number of machines and the initials of the technique to handle the repeated genetic material that it uses as EGGA GE-One machine, EGGA IE-One machine, EGGA GE-Two machine, and EGGA IE-Two machine. For a fair comparison, the four EGGA variants were run with the parametrization described in Section: 4.5. Population size $|P| = 100$; number of individuals selected for the crossover $n_c = 20$; number of individuals selected for the mutation $n_m = 83$; elite population size $|B| = 20$; and, maximal number of generations $max\_gen = 500$. Likewise, we compare the performance of the EGGAs based on their RPD to CPLEX. Table 6.9 shows the experimental results of this phase. The first two columns indicate the criteria used to group the instances, i.e., n, m, p_{ij}, and the 1400

Table 6.9: Performance comparison of the metaheuristic algorithms: EGGA GE-One machine, EGGA GI-One Machine, EGGA GE-Two machines, and EGGA GI-Two Machines based on the average RPD.

Instance Set		EGGA IE-One machine	EGGA GI-One machine	EGGA IE-Two machines	EGGA GI-Two machines
n	100	0.098	0.099	**0.055**	0.099
	200	0.099	0.101	**0.055**	0.101
	500	0.101	0.102	**0.055**	0.102
	1000	0.111	0.114	**0.056**	0.114
m	10	0.109	0.111	**0.056**	0.111
	20	0.107	0.108	**0.056**	0.108
	30	0.105	0.106	**0.056**	0.106
	40	0.104	0.104	**0.057**	0.104
	50	0.102	0.102	**0.057**	0.102
p_{ij}	$U(1, 100)$	0.206	0.210	**0.076**	0.210
	$U(10, 100)$	0.191	0.203	**0.076**	0.203
	$U(100, 120)$	0.035	0.036	**0.023**	0.036
	$U(100, 200)$	0.117	0.122	**0.071**	0.122
	$U(1000, 1100)$	0.016	0.015	**0.011**	0.015
	$JobsCorr$	0.083	0.083	**0.062**	0.083
	$MacsCorr$	0.122	0.111	**0.084**	0.111
1400 instances		0.110	0.112	**0.058**	0.112

instances together. Thus, the remaining columns contain the average RPD obtained by each crossover operator for each grouping criterion, highlighting in bold the best results.

From Table 6.9 can be observed that neither the machines nor the jobs impact the difficulty of the instances, but the criteria used to generate the processing times p_{ij} of the instances do. Thus, instances with generated processing times in the ranges $U(1, 100)$, $U(10, 100)$ and with correlated machines $MacsCorr$ represent the biggest solution challenge. In addition, this table indicates that the crossover operator that provides the best performance is IE-Two machines. Therefore, it will be the operator of the current version of the EGGA. Finally, the results in Table 6.9 showed that this study allowed an improvement rate of about 17%. It is important to highlight that, although the results achieved by the EGGA with the new crossover operator are still far from the best state-of-the-art algorithms, this study allowed knowing in detail the optimization process of the crossover operator for the problem $R||C_{max}$. Such knowledge of the problem domain will be of great use in the following stages of this research project.

6.4 Conclusions of the analysis

In this chapter, we analyzed the performance of the most representative state-of-the-art crossover operators to solve grouping problems that affect the solutions at the group-level. Furthermore, we studied different aspects that intervene during the crossover process, such as the arrangement of the machines in parents, the criteria to establish the machine transmission order, the suitability of generating one or two children, and two strategies to handle the repeated genetic material. We proposed a systematical experimental design based on phases. We started from a completely random crossover operator. In each stage, we proposed different crossover operators to study every aspect mentioned above to try to understand their algorithmic behavior. The knowledge gained from each phase was used to design a specific-purpose crossover operator for $R||C_{max}$, referred to as IE-Two machines. The designed crossover operator was incorporated into the GGA presented in Chapter 4, given rise to the Enhanced GGA (EGGA). The experimental results showed that the systematical experimental study presented in this chapter allowed an improvement rate of about 17%. It is important to note that although this operator did not significantly improve the obtained results, it allowed understanding in detail the way in which the different processes that are part of a crossover operator can impact its performance. In this way, we concluded that the most suitable crossover operator for the problem $R||C_{max}$ should organize the parent machines based on their processing times C_i, rearranging the tied machines according to the number of jobs assigned to them N_{jobs}. During the transmission, both parents must transmit their machines, always giving priority to the one with the lowest C_i and, if necessary, breaking the tie randomly. Finally, the operator must handle the repeated genetic material with the Item Elimination strategy and must generate only one child. In the next chapter, we will analyze the optimization process of the mutation operator with a systematical experimental study similar to the one presented in this chapter in order to create a specific-purpose grouping mutation operator for $R||C_{max}$.

Chapter 7

Mutation operators

After the crossover, the mutation operator is the second most used variation operator in the GGA. Commonly, mutation operators promote the exploration of the search space by slightly altering the solution genetic material. Usually, the mutation behavior is useful for a GGA when it is converging to a local optimum since it provides the capacity to redirect the search to other areas of the search space. Therefore, one of the main challenges during the design of an efficient GGA is the development of an efficient mutation operator. According to Quiroz-Castellanos *et al.*, the performance of the GGA-CGT, the base of the GGA introduced in this work, is mainly related to the mutation operator, which alone is capable of finding quality solutions. Section 4.6 includes an experimental study with different parameter configurations, that allows observing how the performance of the GGA case study of this work is mainly related to the crossover operator, while the mutation operator has a low impact.

The above motivates this work that aims to study the performance of different grouping mutation operators to identify the strategies that they use and that positively impact their performance. In this order of ideas, this chapter presents an experimental design based on phases. In each stage, different strategies are explored to steady how they can affect the performance of a mutation operator. The best strategies will be employed to design a new operator to incorporate it into the GGA in order to improve its performance when solving $R||C_{max}$. The information gathered gave an overview of the options that exist and served as an inspiration to design a specific-purpose mutation operator for $R||C_{max}$. The designed mutation operator was incorporated into the GGA presented in Chapter 4, achieving an improvement rate of about 52%.

7.1 State-of-the-art grouping mutation operators

Mutation is a genetic operator generally used to control population diversity during the GGA search process. The mutation operators for the GGA are called grouping mutation operators since they work at the group-level. That is, they select g groups using some criterion (such as selecting the best, the worst, or random groups), to slightly modify them employing different operations. According to the scope of the

literature review, the state-of-the-art holds seven mutation operators designed for GGAs in addition to the Download operator. Three of them, the Swap, the Insertion, and the Item Elimination, perform small alterations in the solutions with operations directly applied to some items of the selected groups. In contrast, the remaining operators, called Elimination, Creation, Merge & Split, and Reordering, promote more severe disturbances in solutions since they perform operations involving all the items of the selected groups [376].

The seven mutation operators have been used to solve a wide variety of grouping problems with different conditions and constraints. Due to these differences, mutation operators must be adapted to the characteristics of the problem to be solved. As a result, grouping mutation operators can differ in the criteria they use to select the jobs and machines involved in the mutation operations, the strategies employed to handle the jobs and the selected machines, and the problem-domain heuristics included. The following sections consider the general procedure of four state-of-the-art grouping mutation operators: Swap, Insertion, Elimination, and Merge & Split. They were chosen because this study contemplates the best state-of-the-art mutation operators that are suitable for the $R||C_{max}$ problem, and those unsuitable mutation operators and also those which have not shown a competitive performance were discarded. However, in [376] interested readers can find a more detailed description of the seven mutation operators, as well as a compilation of the mutation operators applied to different grouping problems and the parameter setting approach they use. It is important to note that, besides the Download operator, none of the four mutations described below have been used to solve the $R||C_{max}$ grouping problem. The above motivates this experimental study, whose main objective is to explore the performance of the most used mutation operators now to solve $R||C_{max}$.

7.1.1 The Swap operator

The Swap operator selects two groups, to later pick k items from each selected group and exchange the items from one group to another. Due to its way of working, it can be adapted and used to solve grouping problems with different constraints and conditions. Thanks to this feature, the Swap operator has been used to solve classic problems like Bin Packing [85] as well as new problems such as Maximally Diverse [59].

7.1.2 The Insertion operator

Similar to the Swap operator, the Insertion operator selects two groups, to later pick k items from one selected group, and insert them to the other group. This operator has been used to solve from classic problems such as Graph Coloring [52] to newer problems such as Group Stock Portfolio [83], covering problems with different constraints and conditions [54].

7.1.3 The Elimination operator

The Elimination operator chooses g groups to remove them, release their items, and re-insert them by applying problem-domain heuristics, for example, the heuristic Min() used by the GGA for $R||C_{max}$ (see Chapter 4). According to the scope of the literature review, this is the most used mutation operator to solve grouping problems because it has shown promising results, mainly in classic problems like Bin Packing [11], Cell Formation [25], Multiple Knapsack [64], and Timetabling [51].

7.1.4 The Merge & Split operator

The Merge & Split, also known as Division and Combination operator, works in two phases. In the first stage, it selects two groups and transforms them into a single one. Then, in the second stage, it picks a group to distribute its items between two distinct groups. Merge & Split has been used to solve grouping problems like Cell Formation [26] and Multivariate Micro-aggregation [70].

7.2 Experimental design for the $R||C_{max}$ mutation operators

This section presents the experimental design proposed to analyze the way different elements involved in the mutation process can impact on the performance of grouping mutation operators. The objective of this work is to design an efficient grouping mutation operator that includes the best features identified during the experimentation, to later incorporate it into the GGA presented in Chapter 4 and create the second Enhanced GGA (EGGA). The experimental design consists of four phases. The first stage covers the analysis of the state-of-the-art grouping mutation operators to determine which one has the best performance for $R||C_{max}$. The second phase comprises an exploratory analysis to observe the influence of the number of machines and jobs involved in the mutation operations. The third phase includes the assessment of different machine selection strategies, including biased, random, and mixed approaches. Finally, the fourth phase studies the contribution of distinct rearrangement heuristics based on insertion and swap operations. The main objective of these strategies is to reorganize some jobs of the solutions, applying more complex and expensive processes. Although they involve a computational cost, they are very important when the mutation operator by itself is unable to avoid a local optimum. It is important to note that, to establish the order used to analyze the mutation operator aspects, we gave priority to the general elements like the number of genes to mutate and the way they are selected. Thus, we let the more fine factors for the end, such as the rearrangement of the jobs. The information collected is used to design an efficient grouping mutation operator for $R||C_{max}$.

To analyze the performance of each operator, we generate a population of 100 individuals with the Random min() heuristic, to later mutate them for 500 generations.

For a fair comparison, we use the same seed for each operator. The experiments were conducted as follows. First, each mutation operator is applied to the 1,400 test instances introduced by Fanjul-Peyro in 2010. Later, the performance of each mutation operator is calculated based on the RPD measure, presented in Equation 3.6. Finally, for a comprehensive analysis, the performance of the mutation operators is compared with the 1400 instances grouped with four criteria: the number of jobs n, the number of machines m, the distribution of the processing times p_{ij}, and the 1400 instances together. The experimental results are presented in tables, where the first two columns indicate the criteria used to group the instances, i.e., n, m, p_{ij}, and the complete benchmark. Thus, the remaining columns contain the average RPD obtained by each mutation operator for each grouping criterion, highlighting in bold the best results.

The following sections (1) describe the procedure of the operators studied in each phase, (2) contain a comparison of their performance, and (3) highlight the characteristics that show a positive impact on solving the $R||C_{max}$ problem.

7.2.1 State-of-the-art operators

This experiment aims to study the optimization process of the state-of-the-art grouping mutation operators in the problem $R||C_{max}$. This study comprises four operators: Swap, Insertion, Elimination, and Merge & Split. Figure 7.1 presents the procedure of the four mutation operators adapted to work with the constraints and conditions of the $R||C_{max}$ problem.

The Swap operator selects two jobs j_A and j_B from two different machines i_A and i_B to interchange them. Both jobs and machines are selected randomly. Figure 7.1a explains the mutation process of the Swap operator adapted to solve the $R||C_{max}$ problem with an example in which jobs $j_A = j_1$ and $j_B = j_7$, selected from machines $i_A = i_1$ and $i_B = i_4$, respectively, are exchanged. In this way, in the initial individual ($Solution$), machines i_A and i_B outlined in bold, and the jobs in bold j_A and j_B depict the machines and the selected jobs, respectively; and the final individual ($Mutation$) shows the jobs in their new position.

Similarly, the Insertion operator chooses one job j_A from one machine i_A to insert it into another machine i_B. In this case, the machines and the jobs are selected randomly. Figure 7.1b describes the mutation process of the Insertion operator implemented to solve the $R||C_{max}$ problem with an example, where job $j_A = j_7$, selected from machine $i_A = i_4$, is inserted into machine $i_B = i_1$. For a clear explanation, the example outlines in bold the selected machines i_A and i_B and highlights the inserted item in bold j_A in the initial individual ($Solution$). Thus, the final individual ($Mutation$) shows the picked job j_A in its new position.

On the other hand, the Elimination operator randomly selects two machines i_A and i_B and although it does not eliminate them due to the characteristics of the problem, it releases all their jobs to later permute them and re-insert them with the Min() heuristic. Figure 7.1c explains the mutation process of the Elimination operator adapted to solve the $R||C_{max}$ problem with an example, where the machines outlined in bold $i_A = i_3$ and $i_B = i_4$ depict the machines selected to remove their jobs j_3, j_5, j_6, j_7 and j_8,

highlighted in bold from the initial individual (*Solution*). The *Incomplete Solution* shows the chromosome without the released items, placed in the box RJ. Lastly, the box *Permutation* represents the jobs in RJ reordered randomly, and the final solution *Mutation* depicts the chromosome generated by assigning the jobs in the box *Permutation* by using the problem-domain Min() heuristic.

Finally, like Elimination, the Merge & Split operator selects two machines i_A and i_B in a random way, and even it cannot join them, it extracts their jobs and merges them in the set $i_A \cup i_B$. Next, it simulates the split part by re-inserting the released jobs among the two selected machines i_A and i_B using the described-above Best() heuristic. Figure 7.1d includes the mutation process of the Merge & Split operator with an example that contains an initial individual (*Solution*) with the two selected machines i_A and i_B outlined in bold and the released jobs j_3, j_5, j_6, j_7, and j_8 highlighted in bold. Besides, the example contains the *Incomplete Solution* without the jobs in $i_A \cup i_B$, placed in a box with the same name ($i_A \cup i_B$). Lastly, this figure includes the final solution *Mutation* that depicts the chromosome resulted from the allocation of the jobs in *Permutation* (a box with the jobs in $i_A \cup i_B$ reordered randomly) by applying the problem-domain Best() heuristic.

Table 7.1 shows the results obtained in the experiment. For a comprehensive study, the performance of the operators was analyzed considering the four criteria described in the experimental design, i.e., the number of jobs n, the number of machines m, and the distribution of the instances processing times p_{ij}. In this way, the first column indicates the criterion used to study the performance of the operators, the second one contains the classes covered for each grouping criterion, and the following columns represent the average RPD (Relative Percentage Deviation) achieved by each operator: Swap, Insertion, Merge & Split, and Elimination. The last row of the table shows the average RPD of each operator for the 1,400 test instances. From Table 7.1 it can be observed that the Elimination operator excelled in all the criteria used to distribute the instances. It is important to note that the four operators had a similar performance since their average RPD differs only by hundredths.

Moreover, it is remarkable that the Download mutation operator procedure of the GGA presented in Chapter 4 is quite similar to the state-of-the-art Merge & Split mutation operator, since although the operations merge & split cannot be applied to groups explicitly; due to the characteristics and conditions of the problem, they can be emulated by considering the jobs. In this way, the first stage of the Download mutation operator represents the combination of the groups, where the jobs of the two selected machines are released and placed in a single set. Similarly, the second stage depicts the split operation, where the jobs are redistributed among the selected machines. Finally, it is also important to mention that the only difference between the Merge & Split operator and the Elimination operator (the two operators with the best performance) is the job reassignment strategy they work with, since Merge & Split re-inserts the jobs only on the two selected machines, while the Elimination operator tries to re-insert the jobs on all the machines.

Derived from these results, the following stages of this experimental study contain the analysis of different aspects involved in the mutation operator with the reassignment heuristic that consider all the machines, such as the number of machines to handle,

Given a test instance of $R||C_{max}$ with n=9 jobs from j_1 to j_9 and m=4 machines from i_1 to i_4 where each cell indicates the processing time p_{ij} that each machine i requires to process every job j, as well as a solution where each gene represents a machine i that contains its assigned jobs and the processing time C_i that it needs to process such jobs:

Test Instance

M \ N	i_1	i_2	i_3	i_4
j_1	20	15	10	12
j_2	10	12	16	18
j_3	18	15	11	10
j_4	15	17	16	11
j_5	10	20	11	19
j_6	20	12	15	16
j_7	11	19	20	13
j_8	12	10	18	20
j_9	15	20	12	11

Solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4	j_5, j_8	j_3, j_6, j_7
C_i	35	29	29	39

The four group-oriented mutation operators work as follows:

a) Swap

Solution

Machines	i_1	i_2	i_3	i_4
Jobs	$\mathbf{j_1}, j_9$	j_2, j_4	j_5, j_8	$j_3, j_6, \mathbf{j_7}$
C_i	35	29	29	39
	i_A			i_B

Mutation

Machines	i_1	i_2	i_3	i_4
Jobs	j_7, j_9	j_2, j_4	j_5, j_8	j_1, j_3, j_6
C_i	26	29	29	38

b) Insertion

Solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4	j_5, j_8	$j_3, j_6, \mathbf{j_7}$
C_i	35	29	29	39
	i_B			i_A

Mutation

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_7, j_9	j_2, j_4	j_5, j_8	j_3, j_6
C_i	46	29	29	26

c) Elimination

Solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4	$\mathbf{j_5, j_8}$	$\mathbf{j_3, j_6, j_7}$
C_i	35	29	29	39
			i_A	i_B

Incomplete Solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4		
C_i	35	29		

RJ: j_3, j_5, j_6, j_7, j_8

Mutation

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4, j_8	j_5, j_6	j_3, j_7
C_i	35	39	26	23

Permutation: j_6, j_7, j_5, j_3, j_8

d) Merge & Split

Solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4	$\mathbf{j_5, j_8}$	$\mathbf{j_3, j_6, j_7}$
C_i	35	29	29	39
			i_A	i_B

Incomplete Solution

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4		
C_i	35	29		

$i_A \cup i_B$: j_3, j_5, j_6, j_7, j_8

Mutation

Machines	i_1	i_2	i_3	i_4
Jobs	j_1, j_9	j_2, j_4	j_5, j_6, j_8	j_3, j_7
C_i	35	29	44	23

Permutation: j_6, j_7, j_5, j_3, j_8

Figure 7.1: Group-oriented mutation operators adapted for $R||C_{max}$.

the number of jobs to remove, the machine selection strategy, and the rearrangement heuristics.

Table 7.1: Comparison of Swap, Insertion, Merge & Split, and Elimination mutation operators using RPD.

	Instance set	**Swap**	**Insertion**	**Merge & Split**	**Elimination**
n	100	0.1213	0.1219	0.1071	**0.0804**
	200	0.1408	0.1432	0.1353	**0.1154**
	500	0.1365	0.1371	0.1372	**0.1281**
	1000	0.1380	0.1381	0.1387	**0.1350**
m	10	0.1291	0.1290	0.1291	**0.1178**
	20	0.1391	0.1402	0.1344	**0.1229**
	30	0.1256	0.1252	0.1220	**0.1074**
	40	0.1310	0.1331	0.1270	**0.1084**
	50	0.1460	0.1478	0.1353	**0.1172**
P_{ij}	$U(1, 100)$	0.2802	0.2740	0.2632	**0.2107**
	$U(10, 100)$	0.2080	0.2060	0.2039	**0.1802**
	$U(100, 120)$	0.0417	0.0438	0.0408	**0.0384**
	$U(100, 200)$	0.1230	0.1248	0.1198	**0.1164**
	$U(1000, 1100)$	0.0218	0.0230	0.0214	**0.0201**
	$JobsCorr$	0.1259	0.1307	0.1194	**0.1049**
	$MacsCorr$	0.1385	0.1432	0.1384	**0.1326**
	1400 instances	0.1341	0.1351	0.1296	**0.1147**

7.2.2 Handled machines and removed jobs

After observing that the four operators of the state-of-the-art showed quite similar performance and that the Elimination operator slightly excelled, the second phase of the experimental study focused on analyzing how the number of handled machines and removed jobs impact on the performance of the mutation operator. To analyze this phenomenon, we explored thirty-five variants of the operator. This study consists of evaluating the suitability of removing 1, 2, 3, 4, 6, 8, and 10 jobs from 2, 4, 6, 8, and 10 different machines, where each combination of removed jobs and managed machines results in an operator. For a fair comparison, all the operators use randomness to select the machines and the jobs that intervene in their mutation process. Thus, each operator releases k jobs from g machines and then re-insert them with the Min() heuristic. As in the first phase, for each operator, 100 individuals were generated and mutated during 500 generations using the same seed.

Table 7.2 shows the experimental results of the thirty-five variants of the mutation operator. The first column indicates the number of machines that each operator manages, the second one represents the number of jobs removed from each of the

handled machines, and the last column contains the average RPD of each operator for the 1,400 test instances.

It appears from Table 7.2 that the operators that release only one job from each machine perform better than those that release more and that the best option is to consider only two machines. Moreover, to graphically observe the behavior of the thirty-five designed operators, the 1,400 instances were grouped into twenty groups concerning each combination of jobs (100, 200, 500, and 1000) and machines (10, 20, 30, 40, and 50) to calculate the average RPD of each group and analyze the impact of each operator in more detail, e.g., the group where $m = 10$ and $n = 100$, the group where $m = 10$ and $n = 200$, and so on. Figures 7.2 and 7.3 contain two representative graphs of the behavior presented by the thirty-five mutation operator variants, which allow observing the impact of the two evaluated features, i.e., the number of machines to be handled and the number of jobs to be removed from each machine.

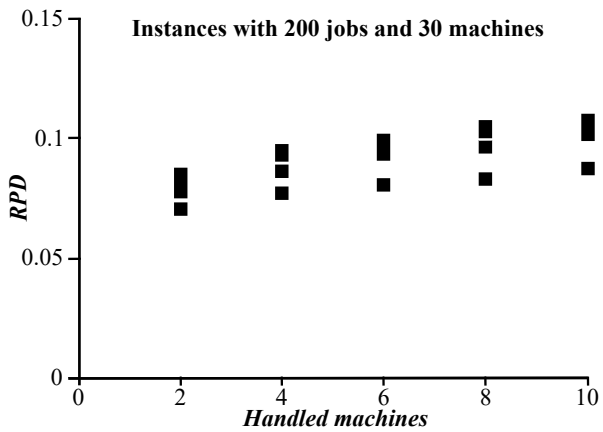


Figure 7.2: Behavior of the mutation operators grouped by the number of handled machines.

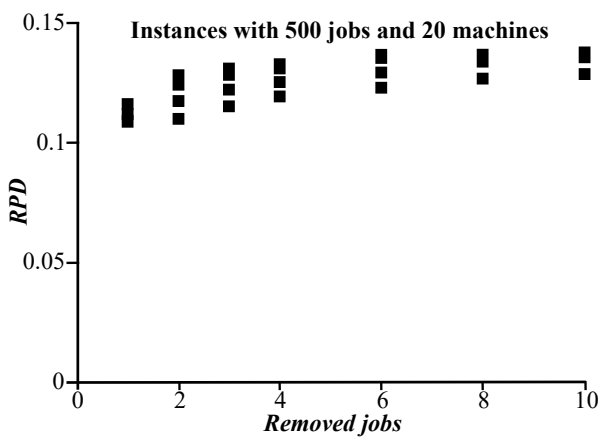


Figure 7.3: Behavior of the mutation operators grouped by the number of removed jobs from the handled machines.

Figure 7.2 exhibits the behavior of the operators, grouped according to the number of machines that they handle for all instances with 200 jobs and 30 machines. The $x-$axis of this figure indicates the number of machines handled, and the $y-$axis contains the

Table 7.2: Comparison of handled machines and removed jobs using RPD.

Handled machines	**Removed jobs**	***RPD***
2	1	**0.091437**
	2	0.094475
	3	0.097644
	4	0.100010
	6	0.102259
	8	0.103456
	10	0.103984
4	1	0.093067
	2	0.100647
	3	0.104505
	4	0.107246
	6	0.109475
	8	0.111302
	10	0.111263
6	1	0.095776
	2	0.105834
	3	0.109519
	4	0.111925
	6	0.114454
	8	0.115151
	10	0.115754
8	1	0.09889
	2	0.109016
	3	0.112681
	4	0.114861
	6	0.116797
	8	0.117525
	10	0.117800
10	1	0.102228
	2	0.110804
	3	0.114677
	4	0.116184
	6	0.117627
	8	0.118031
	10	0.117819

average RPD reached for each operator. On the other hand, Figure 7.3 groups the operators according to the number of jobs removed from each machine in instances with 500 jobs and 20 machines. The $x-$axis contains the operators grouped according to the number of jobs that they remove, and the $y-$axis contains the average RPD reached for each operator. In this way, Figure 7.2 suggests that the performance of the operators improves as the number of handled machines decreases, while Figure 7.3 shows that the operators removing fewer jobs have better performance. In this way, the analysis indicates that those operators handling a fewer number of machines and releasing fewer jobs are more suitable.

7.2.3 Machines selection strategy

Once identified that the variant that considers two machines and releasing one job from each machine has the best performance, in this stage we evaluate the performance of four machine selection strategies: Random, Worst, Worst Best, and Worst Random, so as to analyze how they affect the performance of the mutation operators. Given a solution to be mutated, these strategies work as follows. The Random strategy chooses the two machines precisely at random. The Worst strategy selects the two machines with the worst C_i values (i.e., those machines with the highest loads). The Worst Best strategy picks the worst and the best machine (i.e., the machines with the highest and the lowest C_i values). If there are several machines with the lowest or highest load, as a first step, they are identified to later use a uniform distribution to select one of them randomly. Finally, the Worst Random strategy divides the machines into two groups (W and O), in such a way that W contains the machines with $C_i = C_{max}$ and O the remaining machines. Next, it randomly selects the machines w and o from sets W and O, respectively. It is important to note that for each machine selection strategy, the two released jobs are selected randomly and later re-inserted using the Min() heuristic.

Table 7.3 shows the experimental results of the operators with the four machine selection strategies. As can be seen, this table has the same structure as Table 7.1. That is, it clusters the instances according to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij} of the instances, and the 1,400 test instances together. Therefore, the first column indicates the criterion used to study the performance of the operators, the second one contains the classes covered for each grouping criterion, and the following columns represent the average RPD (Relative Percentage Deviation) achieved by the operators with each machine selection strategy: Random, Worst, Worst Best, and Worst Random. The experimental results in Table 7.3 suggest that the most suitable machine selection strategy is Worst Random, with an average RPD of 0.0674 since the other approaches (Random, Worst, and Worst Best) reached higher RPD averages of 0.0913, 0.0875, and 0.0912, respectively.

7.2.4 Rearrangement heuristics

After identifying the machine selection strategy that provides the best performance to the mutation operator, we noted that there are high possibilities that the genetic

Table 7.3: Comparison of mutation operators with Random, Worst, Worst Best, and Worst Random selection strategies using RPD.

	Instance set	**Random**	**Worst**	**Worst Best**	**Worst Random**
n	100	0.0605	0.0577	0.0618	**0.0296**
	200	0.0848	0.0797	0.0832	**0.0533**
	500	0.1030	0.0987	0.1028	**0.0827**
	1000	0.1175	0.1147	0.1178	**0.1046**
m	10	0.0873	0.0857	0.0894	**0.0718**
	20	0.0978	0.0942	0.0977	**0.0752**
	30	0.0842	0.0824	0.0854	**0.0635**
	40	0.0908	0.0853	0.0885	**0.0631**
	50	0.0963	0.0900	0.0951	**0.0634**
P_{ij}	$U(1, 100)$	0.1430	0.1470	0.1522	**0.1146**
	$U(10, 100)$	0.1321	0.1319	0.1362	**0.1003**
	$U(100, 120)$	0.0351	0.0309	0.0329	**0.0244**
	$U(100, 200)$	0.1017	0.0939	0.0970	**0.0740**
	$U(1000, 1100)$	0.0182	0.0155	0.0171	**0.0123**
	$JobsCorr$	0.0909	0.0820	0.0810	**0.0576**
	$MacsCorr$	0.1179	0.1112	0.1220	**0.0888**
	1400 instances	0.0913	0.0875	0.0912	**0.0674**

material of many solutions does not undergo any alteration during the mutation process. Such a phenomenon can occur because it is likely that the two released jobs can be re-inserted to the same machine to which they belonged. In order to analyze the above, we evaluated the success rate (i.e., the number of the alterations in the genetic material divided by the mutation attempts) of the mutation operator with the best properties identified in the two previous stages. The experimental results revealed that only about the 42% of the mutation attempts are successful. The above motivates this stage of the experimental study that consists in evaluating the utility of incorporating two rearrangement heuristics, called Insertion and Assemble, to increase the operator's success rate and improve its performance. These heuristics are only used if, after releasing and re-inserting the jobs, the genetic material of the mutated solution has not been altered. It is important to note that at this stage we tested the performance of different variants based on insertion and interchange operations, but we only kept the rearrangement heuristics that showed the best results.

The Insertion rearrangement heuristic seeks to reduce the number of jobs in one of the two selected machines by trying to insert each of their jobs into the other ones. Algorithm 1 has the Insertion procedure. We denote $S' = \text{Insertion}(S, j_{sm}, sm, i)$ the solution derived from S by inserting job j_{sm} (j_w or j_o) from the selected machine sm (w or o) into machine i. As can be seen, this heuristic goes through the jobs j_w and j_o of the machines w and o selected with the machine selection strategy Worst Random (line

1). Thus, for each pair of jobs (j_w and j_o), this algorithm traverses the m machines (line 2). In this way, for each machine i, different from machine w and o (line 3 and line 9), it tries to insert job j_w of the worst machine w (line 3) and then job j_o from the other machine o (line 7) following two conditions, denoted as Cnd_1 and Cnd_2.

Cnd_1(S, j_{sm}, sm, i) (line 4 and line 10) allows verifying that the mutated solution ($S^{'}$) will have equal or better quality than the initial solution (S). In this way, Cnd_1 checks out that the sum of the processing time resulted from the insertion in the intervened machines i and sm (w or o) will be less than or equal to the sum of their processing times without performing the insertion. Hence, for each job j_w, Cnd_1(S, j_w, w, i) returns TRUE if $C_w - p_{wj_w} + C_i + p_{ij_w} \leq C_w + C_i$, where C_w and C_i represent the time that machines w and i require to process their assigned jobs, respectively; while p_{wj_w} and p_{ij_w} depict the processing time that machines w and i require to process job j_w, respectively. Otherwise, it returns FALSE. In the same way, for each job j_o, Cnd_1(S, j_o, o, i) returns TRUE if $C_o - p_{oj_o} + C_i + p_{ij_o} \leq C_o + C_i$, where C_o and C_i represent the time that machines o and i require to process their assigned jobs, respectively; while p_{oj_o} and p_{ij_o} depict the processing time that machines o and i require to process job j_o, respectively. Otherwise, it returns FALSE.

On the other hand, Cnd_2(S, j_{sm}, sm, i) (line 4 and line 10) checks out that the mutated solution ($S^{'}$) will have equal or better quality than the initial solution (S). Cnd_2 verifies that the processing time C_i of machine i with the new job, either j_w or j_o, will be less than or equal to the current makespan C_{max}. Therefore, for each job j_w, Cnd_2(S, j_w, w, i) returns TRUE if $C_i + p_{ij_w} \leq C_{max}$. Otherwise, it returns FALSE. Similarly, for each job j_o, Cnd_2(S, j_o, o, i) returns TRUE if $C_i + p_{ij_o} \leq C_{max}$. Otherwise, it returns FALSE.

In this way, the Insertion(S, j_{sm}, sm , i) function (lines 5 and 11) is applied to S if and only if a job j (j_w or j_o) satisfies the two conditions (Cnd_1 and Cnd_2). The rearrangement process ends once an insertion is performed (lines 6 and 12) but, if none of the jobs satisfied the two conditions, the mutated solution would remain with its genetic material without any modification.

On the other hand, the Assemble rearrangement heuristic uses two functions. The first one is the Insertion(S, j_{sm}, sm, i) that works similarly to the before-mentioned rearrangement heuristic. Additionally, it incorporates a second function called Interchange that seeks to exchange each job of the selected machines with each job of the other machines in an attempt to reduce the processing time of the selected machines. Algorithm 2 contains the procedure of the Assemble rearrangement heuristic. We denote as $S^{'}$ = Interchange(S, j_{sm}, sm, j_i, i) the solution derived from S by exchanging job j_{sm} (j_w and j_o) from the selected machine sm (w or o) with each job j_i in machine i. Like the Insertion rearrangement heuristic, Assemble loops through jobs j_w and j_o of machines w and o selected with the Worst Random machine selection strategy (line 1). Thus, for each pair of jobs (j_w and j_o), this algorithm goes through the m machines (line 2). In this fashion, first, it tries to insert jobs j_w of the worst machine w and j_o of the other machine o into every machine i different from machines w and o (line 3 and line 9) according to the two conditions described in Algorithm 1: Cnd_1 and Cnd_2 (line 4 and line 10). Next, it attempts to interchange the same jobs j_w and j_o with each job j_i in every machine i (line 15) different from machine w and o

Algorithm 1 Insertion rearrangement heuristic

Input: A solution S and two machines w and o.
Output: A mutated solution $S^{'}$.

```
1:  for all job j_w ∈ w & j_o ∈ o do
2:      for machine i in S do
3:          if i != w then
4:              if Cnd_1(S, j_w, w, i) and Cnd_2(S, j_w, w, i) then
5:                  S' = Insertion (S, j_w, w, i);
6:                  end process;
7:              end if
8:          end if
9:          if i != o then
10:             if Cnd_1(S, j_o, o, i) and Cnd_2(S, j_o, o, i) then
11:                 S' = Insertion (S, j_o, o, i);
12:                 end process;
13:             end if
14:         end if
15:     end for
16: end for
```

(line 16 and line 22), validating two conditions: Cnd_3 and Cnd_4 (line 17 and line 23).

Cnd_3(S, j_{sm}, sm, j_i, i) (line 17 and line 23) allows verifying that the mutated solution ($S^{'}$) will have equal or better quality than the initial solution (S). In this way, Cnd_3 checks out that the processing time resulted from the exchange in the intervened machines i and sm (w or o) will be less than or equal to the sum of their processing times without swapping their jobs. Hence, for each job j_w, Cnd_3(S, j_w, w, j_i, i) returns TRUE if $(C_w - p_{wj_w} + p_{wj_i}) + (C_i - p_{ij_i} + p_{ij_w}) \leq C_w + C_i$, where C_w and C_i represent the time that machines w and i require to process their assigned jobs, respectively; p_{wj_w} and p_{ij_i} depict the processing time that machines w and i require to process jobs j_w and j_i, respectively; and p_{wj_i} and p_{ij_w} indicate the processing time that machines w and i require to process jobs j_i and j_w, respectively. Otherwise, it returns FALSE. In the same way, for each job j_o, Cnd_3(S, j_o, o, j_i, i) returns TRUE if $(C_o - p_{oj_o} + p_{oj_i}) + (C_i - p_{ij_i} + p_{ij_o}) \leq C_o + C_i$, where C_o and C_i represent the time that machines o and i require to process their assigned jobs, respectively; p_{oj_o} and p_{ij_i} depict the processing time that machines o and i require to process jobs j_o and j_i, respectively; and p_{oj_i} and p_{ij_o} indicate the processing time that machines o and i require to process jobs j_i and j_o, respectively. Otherwise, it returns FALSE.

On the other hand, the condition Cnd_4(S, j_{sm}, sm, j_i, i) (line 17 and line 23) validates that the processing time resulted from the interchange in the intervened machines i and sm (w or o) will be less than or equal to the current makespan (C_{max}) of the initial solution S. Hence, for each job j_w, Cnd_4(S, j_w, w, j_i, i) returns TRUE if $(C_w - p_{wj_w} + p_{wj_i} \leq C_{max})$ and $(C_i - p_{ij_i} + p_{ij_w} \leq C_{max})$. Otherwise, it returns FALSE. Similarly, for each job j_o, Cnd_4(S, j_o, o, j_i, i) returns TRUE if $(C_o - p_{oj_o} + p_{oj_i} \leq$

C_{max}) and ($C_i - p_{ij_i} + p_{ij_o} \leq C_{max}$). Otherwise, it returns FALSE.

The Assemble process ends once an operation, either the insertion or the interchange, is accomplished (lines 6, 12, 19, and 25). If none of the jobs met the conditions, the mutated solution remains with its genetic material without any modification.

Algorithm 2 Assemble rearrangement heuristic

Input: A solution S and two machines w and o.
Output: A mutated solution S'.

```
1:  for all job j_w ∈ w & j_o ∈ o do
2:      for machine i in S do
3:          if i != w then
4:              if Cnd_1(S, j_w, w, i) and Cnd_2(S, j_w, w, i) then
5:                  S' = Insertion (S, j_w, w, i);
6:                  end process;
7:              end if
8:          end if
9:          if i != o then
10:             if Cnd_1(S, j_o, o, i) and Cnd_2(S, j_o, o, i) then
11:                 S' = Insertion (S, j_o, o, i);
12:                 end process;
13:             end if
14:         end if
15:         for job j_i in i do
16:             if i != w then
17:                 if Cnd_3(S, j_w, w, j_i, i) and Cnd_4(S, j_w, w, j_i, i) then
18:                     S' = Interchange (S, j_w, w, j_i, i);
19:                     end process;
20:                 end if
21:             end if
22:             if i != o then
23:                 if Cnd_3(S, j_o, o, j_i, i) and Cnd_4(S, j_o, o, j_i, i) then
24:                     S' = Interchange (S, j_o, o, j_i, i);
25:                     end process;
26:                 end if
27:             end if
28:         end for
29:     end for
30: end for
```

In this way, two variants of the operator with the best characteristics identified in the two previous stages (i.e., removing one job from two machines selected with the Worst Random strategy and re-inserting such jobs with the Min() heuristic) were created, one for each rearrangement heuristic presented in this section: Insertion and Assemble. The performance of the two variants, referred to as Insertion and Assemble, was evaluated using the experimental approach mentioned above, i.e., starting from an initial population of 100 individuals that are subsequently mutated during 500

generations and using the same seed. Table 7.4 holds the experimental results obtained by the two mutation operators generated in this phase. Additionally, Table 7.4 includes the performance of the Download mutation operator, the original GGA operator introduced in Section 4.6, to compare the degree of improvement provided by the variants of the operator proposed in this section. For a comprehensive analysis, the performance of the operators was analyzed by clustering the instances with the criteria used in the previous stages: number of jobs n, number of machines m, distribution of processing times p_{ij}, and the 1400 instances together. Thus, each column shows the performance of each assessed operator for the different criteria used to group the instances.

As it can be observed in Table 7.4, the best variant is that with the Assemble rearrangement heuristic , which for each pair of jobs first tries the insertion and then the interchange. The variants with the Insertion and Assemble rearrangement heuristics reached an average RPD of 0.0552 and 0.0395, respectively. However, it is important to note that the two versions of the mutation operators presented in this section outperformed the original Download mutation operator of the GGA studied, that reached an average RPD of 0.1139, as well as the four state-of-the-art operators, which had an average RPD above 0.1.

Table 7.4: Comparison of mutation operators with the Insertion and Assemble rearrangement heuristics and also the Download operator, using RPD.

	Instance set	**Insertion**	**Assemble**	**Download**
n	100	0.0306	**0.0185**	0.0730
	200	0.0480	**0.0280**	0.1125
	500	0.0631	**0.0441**	0.1328
	1000	0.0793	**0.0671**	0.1383
m	10	0.0612	**0.0416**	0.1261
	20	0.0617	**0.0429**	0.1258
	30	0.0497	**0.0366**	0.1076
	40	0.0507	**0.0376**	0.1054
	50	0.0528	**0.0382**	0.1048
P_{ij}	$U(1,100)$	0.0523	**0.0407**	0.2307
	$U(10,100)$	0.0538	**0.0331**	0.1862
	$U(100,120)$	0.0286	**0.0176**	0.0358
	$U(100,200)$	0.0750	**0.0362**	0.1072
	$U(1000,1100)$	0.0150	**0.0100**	0.0182
	$JobsCorr$	0.0664	**0.0654**	0.0892
	$MacsCorr$	0.0952	**0.0728**	0.1304
	1400 instances	0.0552	**0.0394**	0.1139

7.3 GGA with the old and the new mutation operators

Given the knowledge gained from the experimental study, we propose a mutation operator called 2-Items Reinsertion. This operator randomly chooses two jobs from two different machines selected with the Worst Random strategy to release them and later re-insert them with the Min() allocation heuristic. Furthermore, it employs the Assemble rearrangement heuristic, based on insertion and interchange operations. The rearrangement process is only applied if, after releasing and re-inserting the jobs, the genetic material of the mutated solution has not been modified.

To assess the 2-Items Reinsertion mutation operator performance, we run two variants of the GGA for $R||C_{max}$. The GGA introduced in Chapter 4 that uses the Download operator and the EGGA resulted from the incorporation of the 2-Items Reinsertion mutation into GGA, instead of the Download operator. Both GGAs are evaluated over the 1400 benchmark instances. To promote a fair comparison, the effectiveness and efficiency of both GGA were compared by using the same parameter configuration. Population size $|P| = 100$; number of individuals selected for the crossover $n_c = 20$; number of individuals selected for the mutation $n_m = 83$; elite population size $|B| = 20$; and, maximal number of generations $max\_gen = 500$. In this way, we analyze the strengths and weaknesses of the 2-Items Reinsertion mutation operator, distinguishing the quality of the solutions found by each GGA variant, their search time, as well as their ability to escape from local optima. For a fair comparison, both algorithms were programmed in the Rust language and were compiled using Visual Studio in the 64-bit mode. The experiments were performed on a computer with an Intel Core i5 (3.10 GHz), and 16 GB in RAM. Finally, for each instance, a single execution of the algorithms were run, with the same initial seed for the random number generation.

7.3.1 Comparing the effectiveness of GGA with the old and the new mutation operators

To measure the effectiveness of the designed 2-Items Reinsertion mutation operator, we applied the two GGA variants to the 1400 test instances and measured the improvement degree in the quality of the solutions found by each algorithm based on the RPD. Table 7.5 contains the experimental results. The first and second columns indicate the criteria used to group the test instances, based on the number of jobs n, the number of machines m, the processing time distribution p_{ij}, and the 1400 instances together. On the other hand, the remaining columns contain the average RPD obtained by each metaheuristic algorithm for the four grouping criteria, highlighting in bold the metaheuristic algorithm with the lowest average RPD for each set.

From Table 7.5 can be observed that the EGGA showed a better performance than GGA using any criteria to group the test instances. Furthermore, it is worth noting that the EGGA reaches an average RPD considerably lower than the initial GGA by solving the 1,400 test instances, with 0.028 and 0.059, respectively.

Table 7.5: Comparison of the GGA and the EGGA presented in this chapter, using RPD.

	Instance set	**GGA**	**EGGA**
n	100	0.0659	**0.0176**
	200	0.0655	**0.0224**
	500	0.0657	**0.0291**
	1000	0.0688	**0.0441**
m	10	0.0683	**0.0220**
	20	0.0683	**0.0306**
	30	0.0683	**0.0275**
	40	0.0683	**0.0308**
	50	0.0683	**0.0306**
P_{ij}	$U(1,100)$	0.1027	**0.0465**
	$U(10,100)$	0.1119	**0.0361**
	$U(100,120)$	0.0256	**0.0092**
	$U(100,200)$	0.0829	**0.0229**
	$U(1000,1100)$	0.0121	**0.0036**
	$JobsCorr$	0.0586	**0.0380**
	$MacsCorr$	0.0955	**0.0419**
	1400 instances	0.0699	**0.0283**

Finally, in order to graphically show the suitability of the designed mutation operator, the experimental study presented in Section 4.6 was repeated, but this time for the impact analysis of crossover and mutation rates on the EGGA. In this way, the EGGA was run with the same nine configurations, i.e., $Conf_1$: $n_c = 20$, $n_m = 20$, $Conf_2$: $n_c = 20$, $n_m = 40$, ... $Conf_9$: $n_c = 60$, $n_m = 60$. Figure 7.4 presents a bar graph with the obtained results from this study, where each bar depicts one of the nine configurations grouped according to the number of mutated solutions (n_m), and each pattern indicates the number of selected individuals for the crossover process (n_c): squares = 20, waves= 40, and circles= 60. As Figure 7.4 indicates, the EGGA performance is mainly related to the number of individuals considered for the mutation processes n_m, as the performance of the EGGA improves (lower RPD) as the number of mutated solutions increases. Similarly, as the number of selected individuals for the crossover process n_c increases, the GGA performance improves, but to a lesser degree. The behavior mentioned above shows the suitability of the 2-Items Reinsertion mutation, which is the operator with the biggest impact on EGGA final performance and improves it considerably. Thus, the EGGA behavior is quite similar to the one presented by the GGA-CGT [11], where the mutation operator has the greatest positive impact on the final performance of this algorithm.

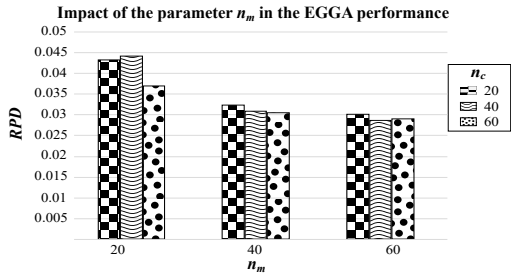


Figure 7.4: Impact analysis of the parameters: number of individuals selected for crossover n_c and number of mutated solutions n_m, in the EGGA performance.

7.3.2 Comparing the efficiency of GGA with the old and the new mutation operators

After analyzing the effectiveness of the EGGA, we evaluate the implications associated with the computational time of using the 2-Items Reinsertion mutation operator. Table 7.6 includes the experimental results. Like Table 7.5, the first and second columns describe the characteristics used to cluster the instances: the number of jobs n and machines m, the processing time distribution p_{ij}, and the 1400 instances together. The following columns contain the average time in seconds obtained by the GGA and the EGGA for each instance set, respectively, highlighting in bold the metaheuristic algorithm with the lowest computational cost for each set.

Table 7.6: Comparison of the GGA and the EGGA based on time (in seconds).

	Instance	**GGA**	**EGGA**
n	100	**1.2**	5.71
	200	**1.2**	5.68
	500	**1.24**	5.49
	1000	**1.36**	9.44
m	10	**1.26**	8.71
	20	**1.24**	7.66
	30	**1.21**	6.94
	40	**1.19**	6.33
	50	**1.17**	5.79
P_{ij}	$U(1,100)$	**1.25**	34.09
	$U(10,100)$	**1.25**	14.04
	$U(100,120)$	**1.25**	2.52
	$U(100,200)$	**1.25**	2.88
	$U(1000,1100)$	**1.25**	2.71
	$JobsCorr$	**1.25**	1.50
	$MacsCorr$	**1.25**	1.69
	1400 instances	**1.25**	8.49

From Table 7.6 can be concluded that the 2-Items Reinsertion mutation operator causes the EGGA to be much slower. Said computational cost is closely related to the Assemble rearrangement strategy, incorporated to avoid, as far as possible, getting stuck in a local optima. This effect is mainly observed in those instances with processing times generated in the ranges $U(1, 100)$ and $U(10, 100)$, where the average times increased from 1.25 to 34.09 and 14.04 seconds, respectively. Although the computational cost of the Assemble rearrangement strategy is high, it is also too useful, since the properties and characteristics of the addressed problem make the mutation operator by itself incapable of avoiding local optima. Table 7.6 also indicates that the EGGA is approximately eight times slower than the initial GGA. Therefore, to make a fair comparison and show the usefulness of the Assemble heuristic, we execute the original GGA increasing the number of generations eight times, that is, 8 * 500 = 4000 generations. Table 7.7 includes the experimental results.

Table 7.7: Performance analysis of the GGA with 500 and 4000 generations, and the EGGA with 500 generations.

	Instance	**GGA (500 gen)**	**GGA (4000 gen)**	**EGGA (500 gen)**
n	100	0.0659	0.0571	**0.0176**
	200	0.0655	0.0564	**0.0224**
	500	0.0657	0.0559	**0.0291**
	1000	0.0688	0.0600	**0.0441**
m	10	0.0683	0.0592	**0.0220**
	20	0.0683	0.0592	**0.0306**
	30	0.0683	0.0590	**0.0275**
	40	0.0683	0.0588	**0.0308**
	50	0.0683	0.0585	**0.0306**
P_{ij}	$U(1, 100)$	0.1027	0.0934	**0.0465**
	$U(10, 100)$	0.1119	0.0979	**0.0361**
	$U(100, 120)$	0.0256	0.0215	**0.0092**
	$U(100, 200)$	0.0829	0.0719	**0.0229**
	$U(1000, 1100)$	0.0121	0.0101	**0.0036**
	$JobsCorr$	0.0586	0.0458	**0.0380**
	$MacsCorr$	0.0955	0.0812	**0.0419**
	1400 instances	0.699	0.0603	**0.0283**

Table 7.7 allows observing that the GGA cannot reach the EGGA performance, not even increasing the number of generations eight times to make a fair comparison with respect to search time. This behavior can be related to the explorations and exploitation capabilities of the initial GGA, which cause it to stagnate in local optima. To review such algorithmic behavior, we analyzed the average generation in which the GGA and the EGGA find the best solution for each test instance. Both algorithms were executed using the same parameter configuration, proposed in [7]. Table 7.8 has the experimental results, highlighting in bold the metaheuristic algorithm that avoids getting stuck in

local optima for more generations on average for each set.

Table 7.8: Comparison of the GGA and the EGGA based on the generation in which the best solution in the population is improved.

	Instance	**GGA**	**EGGA**
n	100	9.27	**358.34**
	200	8.00	**369.31**
	500	8.00	**380.83**
	1000	17.09	**362.54**
m	10	16.35	**358.46**
	20	13.69	**359.06**
	30	11.79	**359.78**
	40	11.05	**360.96**
	50	10.01	**362.20**
P_{ij}	$U(1, 100)$	64.80	**218.56**
	$U(10, 100)$	8.00	**305.34**
	$U(100, 120)$	8.00	**360.44**
	$U(100, 200)$	8.00	**391.96**
	$U(1000, 1100)$	8.00	**390.13**
	$JobsCorr$	8.00	**474.82**
	$MacsCorr$	8.00	**392.95**
	1400 instances	16.11	**362.03**

From Table 7.8 can be observed that GGA gets quickly trapped in local optima, in generation 16 on average, while the EGGA shows a better ability to deal with the landscape characteristics of the $R||C_{max}$ search space, finding its best solutions in generation 362 on average. In this way, Table 7.8 remarks the importance of incorporating the 2-Items Reinsertion mutation operator because even requiring a high computational cost, it provides the EGGA a better exploration capability during the search process.

7.4 Conclusions of the analysis

The main goal of this chapter was to promote the design of intelligent operators for GGAs as a more suitable way to obtain high-performance GGAs that incorporate knowledge of the problem-domain. In this order of ideas, we presented a systematic experimental examination to gain insights into the importance of each phase involved in the mutation operator of a GGA designed to solve the Parallel-machine scheduling problem with unrelated machines and makespan minimization ($R||C_{max}$), analyzing whether different strategies actually contribute to the performance of the operator. The overall procedure of a grouping mutation operator for $R||C_{max}$ comprises: (1) selecting one or more machines; (2) selecting one or more jobs from each of the selected

machines; and (3) re-inserting the selected jobs in some of the machines. In order to learn about each of these three algorithmic components, this work covered the analysis of each component in isolation by evaluating different strategies to deal with it. In this way, the study covered the evaluation of four state-of-the-art grouping mutation operators, thirty-five operators with different numbers of machines and jobs handled, four machine selection strategies, and two rearrangement heuristics for the re-insertion of the selected jobs. The experimental results suggested that the mutation operator with the best performance (1) selects two machines, one of the machines with the worst C_i value and one random machine; (2) selects one random job from each of the selected machines; and (3) re-inserts the selected jobs in two stages, first, for each job, each machine is checked in an attempt to insert the job in the machine with the lowest C_i value, second, if the first stage conduces to the original solution, a rearrangement heuristic is applied attempting to reduce the processing time of the selected machines by trying to insert one of their jobs into the other machines or to exchange one of their jobs with one job of the other machines. The knowledge gained from the systematic study was used to design a new grouping mutation operator, called 2-Items Reinsertion, which was incorporated into the GGA introduced in Chapter 4 (replacing the original mutation operator) to solve 1,400 benchmark instances, showing significant differences with an improvement rate of 52%. These results underline the importance of evaluating the performance of the different components of the GGA operators. The study of the final performance obtained by the Enhanced GGA (EGGA) for the $R||C_{max}$ problem remarked the utility of the designed 2-Items Reinsertion operator because even requiring a high computational cost, it also allows reaching solutions that the initial GGA could not achieve even with more search time due to its fast stagnation in local optima. However, the experimental results also indicate that there still are benchmark instances that show a high degree of difficulty; for such instances, the included strategies in the EGGA do not appear to lead to better solutions. As a result, the EGGA is still getting stuck in local optima, although not as soon as the GGA. In the following chapter, we will employ a study similar to the one presented in this chapter to analyze and improve the reproduction technique in order to provide EGGA the capacity to obtain better solutions.

Chapter 8

Reproduction strategies

One of the fundamentals of genetic algorithms is the principle of natural selection proposed by Darwin. Therefore, the reproduction technique implemented, which consists of selection and replacement mechanisms, has a high impact on its performance. These mechanisms always go hand in hand with the variation operators (generally, crossover and mutation), since the selection picks the individuals for the crossover and mutation process; while the replacement establishes how the offspring and mutated solutions are incorporated into the population [382].

In this chapter, we introduce and analyze the performance of different reproduction techniques that include both random and biased strategies in order to identify the best option for the studied GGA. To achieve this goal, we propose an experimental study consisting of two phases. The first stage includes an exploratory analysis of selection and replacement mechanisms, taken from the state-of-the-art; while the second one contains a study of the way in which the strategies to sort the population impact on the performance of a reproduction technique.

It is important to highlight that the selection and replacement mechanisms studied in this chapter are focused on the crossover operator, since as observed in the Chapter 6, the way in which the solutions are selected, cloned, and mutated together with the proposed specific-purpose mutation operator for $R||C_{max}$ has shown a stable algorithmic behavior.

The knowledge gained from this study will be used to get a performance overview of the strategies that can be incorporated into a reproduction technique. The information collected will be used as a guide to design a purpose-built reproduction technique for $R||C_{max}$. Finally, this chapter presents the last version of the Enhanced GGA (EGGA). This algorithm includes the evolutionary scheme of the initial GGA, presented in Chapter 4, but it uses the population initialization strategy: Random min (presented in Chapter 5), the crossover operator: IE-Two machines (introduced in Chapter 6), mutation operator: 2-Items Reinsertion (introduced in Chapter 7), and the reproduction technique designed based on the knowledge generated from the study presented in this section.

8.1 State-of-the-art of reproduction techniques

The reproduction technique is another important GGA component. In its simplest version, the solutions used by the crossover and mutation operators, as well as the way in which the offspring generated are introduced to the population, are randomly selected. However, this variant is not so frequent, because it is not capable of efficiently controlling the selection pressure, which determines the way in which the individuals in the population converge. The selective pressure is considered one of the critical factors in the design of a GGA since it is responsible for establishing how much priority is given to solutions with the best characteristics to be selected. It is important to note that, at one extreme, high selection pressure can stall the search in a local optimum; while at the other side, a low selection pressure can slow the convergence more than necessary to find the optimal solution [383].

An efficient reproduction technique should promote the generation of solutions with better characteristics as the search process progress. The literature includes approaches that consider the best solution, the worst, and mixed approaches. However, the used approach depends on the constraints and characteristics of the problem to solve. To determine the suitability of a solution to be selected, the reproduction technique uses the evaluation function of the problem to differentiate among the individuals according to their fitness. In this way, the selection and replacement of individuals can be carried out in a more controlled way, providing a balanced selection pressure. In this order of ideas, an efficient selection pressure must adequately control the convergence to avoid stagnation in local optima, and at the same time, it must preserve the diversity of the population to avoid premature convergence. To achieve the expected behavior of the algorithm, it is necessary to know in depth the problem, since depending on the properties of the search space, it will be necessary to implement a different technique. Hence, the main challenge in the design of a high-performance reproduction technique is to identify the set of strategies that helps to maintain a trade-off of exploration and exploitation of the search space and configure them with the suitable selection pressure [382]. Next, the most representative state-of-the-art selection and replacement mechanisms used to design reproduction techniques of genetic algorithms are described.

8.1.1 Selection mechanisms

The specialized literature includes different selection mechanisms that can be incorporated into a genetic algorithm according to the conditions and characteristics of the problem to solve. In this work, we only consider the most representative ones, including Random selection, Ranking selection, Tournament selection, and Proportional selection (or Roulette).

Random selection is the simplest. It uses a probability p generated with a uniform distribution to choose the parents used to generate the offspring of each generation. In this selection mechanism, all the individuals have the same probability of being selected as parents. Therefore, on average, it is the most disruptive selection strategy in terms of breaking genetic codes, as it can combine high- and low-quality solutions. In this

sense, a random selection mechanism can be useful in situations where it is necessary to add a lot of diversity [382].

Unlike the random selection, in the Ranking selection, each individual of the population is ranked according to its fitness. In this way, this strategy has better control of the selection process since it picks the solution based on their quality. Once the population has been ordered, the Ranking selection uses a criterion to determine which parents are chosen, like the best solutions, a combination of best and worst solutions, and a combination of best and random solutions, to mention some examples. This criterion can be adapted according to the conditions and characteristics of the problem. Given its behavior, this strategy is useful in situations where the variance of the quality of the solutions is low [382].

On the other hand, the Tournament selection combines randomness with a bit of bias. To select each parent, this mechanism celebrates a competition among k randomly selected individuals, using a uniform distribution. Subsequently, it ranks the k individuals according to their fitness. Finally, it selects the best solution and adds it to the pool of parents. Such competition is repeated until the pool of parents necessary to generate the offspring of the next generation is accomplished. It should be noted that k is a parameter to configure (usually, equal to 2), which promotes a higher selection pressure as its value increases [384].

Finally, the Proportional selection uses as a principle a linear search through a roulette wheel loaded in proportion to the fitness values of each individual. Thus, each individual has a probability of being selected relative to its fitness, which is equal to its fitness divided by the sum of the fitness of all individuals in the current population. This is how each individual is assigned a piece of the roulette wheel, proportional to its fitness. Once the roulette is created, it is spun to randomly select a parent. Each spin, the individual under the wheel marker is selected. It is important to note that this is a mechanism with moderately strong selection pressure since the selection of the individuals with the best fitness is not guaranteed, but they have higher probabilities of being selected than those of the worst fitness. Given this behavior, this mechanism can be noisy, since its operation depends on the variance of fitness in the population. If the quality difference between the best and the worst solutions is very large, it is likely that premature convergence will occur, since the best solutions have a high probability of being selected more than once. On the other hand, if the quality of solutions is very similar, the probability that each individual is selected will be similar, which makes the process a practically random selection [384].

8.1.2 Replacement mechanisms

Unlike selection strategies, replacement strategies are simpler. Among the most used approaches are Random replacement, Worst replacement, and Parent replacement. As its name indicates, the Random replacement uses a probability p, generated with a uniform distribution, to select the individual of the current population to replace by one of the offspring. On the other hand, the Worst replacement, first sorts the population according to the fitness, to later replace the solutions with the worst fitness. Finally,

the Parent replacement strategy substitutes the individuals of the population used as parents with its offspring [382].

8.2 Experimental design for the $R||C_{max}$ reproduction techniques

This section includes the proposed experimental design to analyze how different aspects that take part in the reproduction technique impact on its performance. The main objective of this study is to identify those components that positively impact solving the problem $R||C_{max}$, to design an efficient reproduction technique that improves the performance of the EGGA. As indicated in previous sections, this work focuses on studying the way in which the solutions that intervene during the crossover process are selected and replaced. Therefore, the selection and replacement of the solutions used for the mutation process are maintained as in the initial GGA. That is, the best n_m solutions are selected and mutated. Also, if the solution belongs to the elite group, first, it is cloned, the clone replaces one of the worst solutions, and then it is mutated.

The experimental design is divided into two phases. The first stage covers the analysis of the state-of-the-art selection and replacement strategies for genetic algorithms to determine which combination (selection-replacement) has the best performance for $R||C_{max}$. On the other hand, the second phase comprises an exploratory analysis to examine the influence of strategies implemented to sort the population before applying the selection and replacement. The information collected is used to design an efficient reproduction technique for $R||C_{max}$.

As in this chapter, we analyze the optimization process of the last GGA component, the algorithm resulted from this study will be the last Enhanced GGA (EGGA). Therefore, the performance of each reproduction technique is assessed on the EGGA with the population initialization strategy: Random min (presented in Chapter 5), the crossover operator (introduced in Chapter 6), and the mutation operators: 2-Items reinsertion (presented in Chapter 7). For a fair comparison, for each reproduction technique studied, we used the following parameter settings. Population size pop_size: 100, number of parents used for the crossover n_c: 40, number of mutated individuals n_m, size of the elite population $|B|$:0.2, maximum number of generations that a solution can be in the population without being modified $lifespan$:10, and maximum number of generations $max\ _gen$: 500. As in the design of the crossover and mutation operators, the performance assessment of each reproduction technique covers the resolution of the 1,400 test instances introduced by Fanjul-Peyro in 2010. Likewise, the performance of the reproduction techniques is compared based on their RPD to CPLEX, represented in Equation 3.6. Finally, we analyze the experimental results by using comparative tables. For a comprehensive analysis of the reproduction techniques algorithmic behavior, tables present the results with the instances grouped according to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij}, and the 1400 instances together.

The following sections describe the procedure of the reproduction techniques studied in

each phase, contain a comparison of their performance, and highlight the characteristics that show a positive impact on solving the $R||C_{max}$ problem.

8.2.1 Selection and Replacement Mechanisms

As indicated in the experimental design, we only consider selection strategies for the crossover operator, as we observed that the selection and replacement strategies of the mutation operator work well. In this order of ideas, in this section, we analyze the algorithm performance of four strategies to select the solutions used as parents during the crossover process, known as Random, Ranking, Tournament, and Proportional, as well as three strategies to replace the offspring generated from the crossover process, referred to as Random, Worst, and Parents. In this way, we studied twelve reproduction techniques that consist of a selection and a replacement strategy. Thus, we use the template *selection replacement* to refer to each reproduction technique studied. The way in which the selection and replacement mechanisms used to create the reproduction techniques were adapted for $R||C_{max}$ are described below.

Random selection is the simplest of the four mechanisms studied. Algorithm 3 contains the procedure followed by this selection mechanism. It starts with a loop that performs n_c iterations, where n_c is the number of children to generate (line 1). Each iteration, it uses the random_selection() method that receives the *population* as a parameter, and it chooses the individual *ind* randomly using a uniform distribution between 1 and the population size *pop_size* (line 2). In this way, the pool *parents* is built by adding the individual in position *ind* in the current population (line 3).

Algorithm 3 Random selection

Input: the current *population*.
Output: the pool with the selected *parents*.

1: **for** *ind* from 1 to n_c **do**
2: *ind* = random_selection(*population*);
3: Append the individual *population*[*ind*] to the pool of *parents*;
4: **end for**

As its name implies, the Ranking selection mechanism uses a strategy to order the population that allows it to pick the parents according to their fitness. Algorithm 4 shows the procedure of this selection mechanism. It starts using the sort() method that receives the *population* as a parameter and returns the *sorted_population* from best to worst, according to fitness. Furthermore, if the solutions have the same C_{max}, the sort() method rearranges the individuals according to their number of machines with a value of $C_i = C_{max}$, from lowest to highest (line 1). Recalling from Chapter 3, C_i indicates the processing time of the machine i and C_{max} represents the maximum C_i in a solution. Subsequently, it iterates from 1 to the number of children to generate n_c (line 2). Finally, in each iteration, the Ranking selection strategy chooses the individual in position *ind* in the *sorted_population* to add it to the pool of *parents* (line 3).

On the other hand, the Tournament selection gives preference to good solutions, avoiding the arrangement of the entire population. Algorithm 5 shows the procedure

Algorithm 4 Ranking selection

Input: the current *population*.
Output: the pool with the selected *parents*.

1: *sorted_population*= sort(*population*);
2: **for** *ind* from 1 to n_c **do**
3: Append the individual *sorted_population*[*ind*] to the pool of *parents*;
4: **end for**

of the implemented selection mechanism. The Tournament selection uses a loop to control the number of tournaments to perform based on the number of children to generate n_c (line 1). Recalling from Chapter 3, C_i indicates the processing time of the machine i and C_{max} represents the maximum C_i in a solution. Each iteration, it uses the random_selection(*population*) method, which receives the *population* as a parameter and returns a randomly chosen individual. In this case, the tournament is celebrated among two individuals: ind_1 and ind_2 (lines 2 and 3). If the first individual selected ind_1 is better (it has a lower C_{max} value) than the second one (line 4), the first individual ind_1 is added to the pool of *parents* (line 5). Otherwise, the second individual ind_2 (line 7) is appended. The process is repeated until the pool of *parents* is accomplished.

Algorithm 5 Tournament selection

Input: the current *population*.
Output: the pool with the selected *parents*.

1: **for** *ind* from 1 to n_c **do**
2: ind_1 = random_selection(*population*);
3: ind_2 = random_selection(*population*);
4: **if** $C_{max}[ind_1] < C_{max}[ind_2]$ **then**
5: Append the individual *population*[ind_1] to the pool of *parents*;
6: **else**
7: Append the individual *population*[ind_2] to the pool of *parents*;
8: **end if**
9: **end for**

Like the Tournament selection, the Roulette selection combines the randomness with the bias to choose the individuals who will participate in the crossover process. Algorithm 6 contains the procedure of the Roulette selection strategy implemented. The algorithm starts using the sort() method that receives the *population* as a parameter and returns the *sorted_population* from the best to the worst, considering the same criteria as in the Ranking selection, i.e., based on their fitness and breaking ties of solutions with the same C_i, according to their number of machines with a value of $C_i = C_{max}$, from lowest to highest (line 1). It then sums the fitness of all the solutions, represented as the $general\_fitness$ (lines 1-4). Next, it creates the vector *probabilities* to save the likelihood of selecting each individual (line 5). Thus, for each individual *ind* in the *population* (line 6), the *probability* of each individual *ind* (line 7) is calculated and added to the vector of *probabilities* (line 8). In this way, the roulette wheel

is constructed with a portion for each individual relative to their fitness. Finally, the Roulette selection uses a loop to select each parent (line 10), by applying the random_selection() method to generate a random probability *rand_prob* between 0 and 1 (line 11). Next, the *probabilities* vector is traversed (line 12) until the individual to be selected is identified (line 15) and added to the pool of *parents* (line 14).

Algorithm 6 Proportional selection

Input: the current *population* and their fitness C_{max}.
Output: the pool with the selected *parents*.

```
1:  sorted_population = sort(population);
2:  for ind from 1 to pop_size do
3:      general_fitness=general_fitness+C_max[ind];
4:  end for
5:  probabilities[];
6:  for ind from 1 to pop_size do
7:      probability=C_max[ind]/general_fitness;
8:      Append the probability to the probabilities vector;
9:  end for
10: for ind from 1 to n_c do
11:     rand_prob= random_selection();
12:     for ind from 1 to pop_size do
13:         if probabilities[ind] ≤ rand_prob then
14:             Append the individual sorted_population[ind] to the pool of parents;
15:             break;
16:         end if
17:     end for
18: end for
```

Similarly, the Replacement mechanisms are implemented using methods alike to those used in Selection mechanisms. The operation of the implemented algorithms is described below. Algorithm 7 shows the procedure of the Random replacement algorithm. For each *child* in the offspring pool generated by the crossover operator (line 1), this replacement strategy uses the random_selection() method to randomly select an individual *ind* from the current population (line 2). Finally, it replaces the individual *ind* selected by the *child* (line 3). Thus, the Random replacement procedure ends by returning the *population* updated.

Algorithm 7 Random replacement

Input: the current *population* and the *offspring*.
Output: the *population* updated.

```
1: for child in offspring do
2:     ind = random_selection(population);
3:     population[ind] = child;
4: end for
```

On the other hand, the Worst replacement sets aside randomness to make use of the

ranking of the solutions and uses the worst criteria to select the individuals to be replaced from the current population. Algorithm 8 shows the Worst replacement procedure. The algorithm starts from the population arranged based on the aforementioned sort() method (line 1), which receives the *population* as a parameter and returns the *sorted_population* from best to worst, according to their fitness. Furthermore, if the solutions have the same C_{max}, the sort() method rearranges the individuals according to their number of machines with a value of $C_i = C_{max}$, from the lowest to the highest. Recalling from Chapter 3, C_i indicates the processing time of the machine i and C_{max} represents the maximum C_i in a solution. Also, this strategy uses the index *ind*, initialized with the population size *pop_size* (line 2) to control the replacement of the worst individuals, placed at the end of the *sorted_population*. Later, it uses a loop to iterate over the children generated by the crossover operator (line 3). At each iteration, the individual in position *ind* of the *sorted_population* is replaced by a *child* of the *offspring* pool (line 4). Finally, in each iteration, the *ind* value is updated by subtracting 1 from it. The Worst replacement procedure ends by returning the *population* updated.

Algorithm 8 Worst replacement

Input: the current *population* and the *offspring*.
Output: the *population* updated.

1: *sorted_population*= sort(population);
2: *ind*=*pop_size*;
3: **for** *child* in *offspring* **do**
4: *sorted_population*[*ind*] = *child*;
5: *ind*–;
6: **end for**

Finally, the Parent replacement uses the randomness to choose some individuals selected as parents to apply the crossover operator to replace them with the offspring. Algorithm 9 contains the procedure of the implemented Parent replacement strategy. The algorithm receives as input the current *population*, the *parents*, and the *offspring*. In this way, it starts by going through each *child* of the *offspring* pool (line 1). Each iteration uses the random_selection() method that receives as a parameter the set of *parents* and returns the individual to replace *ind* (line 2). Finally, it replaces the individual *ind* of the *population* with the *child* in turn. The procedure ends by returning the *population* updated.

Algorithm 9 Parent replacement

Input: the current *population*, the *parents* and the *offspring*.
Output: the *population* updated.

1: **for** *child* in *offspring* **do**
2: *ind* = random_selection(*parents*);
3: *population*[*ind*] = *child*;
4: **end for**

Tables 8.1-8.4 show the *RPD* values obtained by the twelve studied reproduction

techniques, one for each combination of mechanisms of selection and replacement. Each table (8.1, 8.2, 8.3, and 8.4) contains three reproduction techniques with the same selection strategy, either Random, Ranking, Tournament, or Proportional. Thus, each column indicates with which of the three replacement strategies they were combined: Random, Worst, or Parents. For a comprehensive analysis, we compare the performance of the reproduction techniques using the before-mentioned criteria to group the 1400 instances, (1) the number of machines (m), (2) the number of jobs (n), the distribution of the processing times (p_{ij}), and (4) the 1400 instances together.

Table 8.1: Comparison of the reproduction techniques: Random-Random, Random-Parents, and Random-Worst using RPD.

Instance Set		**Random Random**	**Random Parents**	**Random Worst**
n	100	0.034	0.033	**0.027**
	200	0.033	0.033	**0.026**
	500	0.034	0.033	**0.026**
	1000	0.035	0.035	**0.028**
m	10	0.034	0.033	**0.027**
	20	0.034	0.034	**0.027**
	30	0.034	0.034	**0.028**
	40	0.035	0.034	**0.028**
	50	0.034	0.034	**0.028**
p_{ij}	$U(1, 100)$	0.045	0.045	**0.043**
	$U(10, 100)$	0.038	0.038	**0.036**
	$U(100, 120)$	0.011	0.010	**0.009**
	$U(100, 200)$	0.026	0.025	**0.023**
	$U(1000, 1100)$	0.005	0.005	**0.004**
	$JobsCorre$	0.071	0.068	**0.039**
	$MacsCorre$	0.046	0.046	**0.039**
1400 instances		0.035	0.034	**0.028**

From Tables 8.1-8.4 can be concluded that the Worst replacement strategy showed the best performance in all the selection mechanisms studied (Random, Ranking, Tournament, and Proportional). In addition, Table 8.2 suggests that the reproduction technique with the best performance chooses the best individuals of the current population for the crossover process and replaces the worst individuals in the current population with the generated offspring. Given this behavior, in the next phase, the Best-Worst reproduction technique is used, and different ways of organizing the population before applying these selection and replacement strategies are explored.

Table 8.2: Comparison of the reproduction techniques: Ranking-Random, Ranking-Parents, and Ranking-Worst using RPD.

Instance Set		**Ranking Random**	**Ranking Parents**	**Ranking Worst**
n	100	0.027	0.032	**0.025**
	200	0.026	0.032	**0.024**
	500	0.025	0.031	**0.023**
	1000	0.029	0.033	**0.026**
m	10	0.028	0.032	**0.025**
	20	0.028	0.032	**0.025**
	30	0.028	0.033	**0.026**
	40	0.028	0.033	**0.025**
	50	0.028	0.033	**0.025**
p_{ij}	$U(1, 100)$	0.052	0.047	**0.045**
	$U(10, 100)$	0.037	0.038	**0.035**
	$U(100, 120)$	**0.009**	0.010	**0.009**
	$U(100, 200)$	0.021	0.024	**0.020**
	$U(1000, 1100)$	0.004	0.004	**0.003**
	$JobsCorre$	0.036	0.062	**0.032**
	$MacsCorre$	0.038	0.045	**0.035**
1400 instances		0.028	0.033	**0.026**

8.2.2 Strategies to sort the population

After observing that the EGGA works better using the reproduction technique Ranking-Worst, which incorporates a bias towards the selection of the best solutions and the replacement of the worst solutions, in this stage, we will analyze how different strategies to sort the population can affect its performance. The objective of this phase is to identify the repeated solutions to place them at the end of the ordered population. Therefore, the proposed strategies differ in the criteria used to identify repeated solutions. Given the Ranking-Worst procedure, these strategies avoid the selection of the repeated solutions as parents and promote the replacement of the repeated solutions first. It is important to note that the selection mechanism for the mutation process is also affected by the sort strategies studied in this phase, since the mutated solutions are also selected by ranking (the best ones). With this change, the EGGA has more control of the elite population, avoiding repeated solutions. Thus, the probability of stagnation at a local optimum is decreased.

The reproduction techniques studied in this stage work as follows. First, they rank the population according to fitness, to later rearrange the population using the criteria: C_{max}, C_{max} - Machines(C_i=C_{max}), C_{max} - Average(C_i), and C_{max} - Machines(C_i=C_{max}) - Average(C_i). Each strategy is identified with the criterion that they use to rearrange

Table 8.3: Comparison of the reproduction techniques: Tournament-Random, Tournament-Parents, and Tournament-Worst using RPD.

Instance Set		**Tournament Random**	**Tournament Parents**	**Tournament Worst**
n	100	0.031	0.037	**0.026**
	200	0.030	0.037	**0.025**
	500	0.030	0.037	**0.025**
	1000	0.032	0.038	**0.027**
m	10	0.031	0.037	**0.026**
	20	0.031	0.037	**0.026**
	30	0.031	0.037	**0.026**
	40	0.031	0.038	**0.026**
	50	0.031	0.038	**0.026**
p_{ij}	$U(1, 100)$	0.047	0.044	**0.043**
	$U(10, 100)$	**0.036**	0.039	**0.036**
	$U(100, 120)$	0.010	0.011	**0.009**
	$U(100, 200)$	0.023	0.028	**0.022**
	$U(1000, 1100)$	**0.004**	0.005	**0.004**
	$JobsCorre$	0.057	0.087	**0.036**
	$MacsCorre$	0.042	0.050	**0.038**
1400 instances		0.031	0.038	**0.027**

the population. The C_{max} reproduction technique is the simplest since it considered that all the solutions with the same C_{max} are repeated. On the other hand, the C_{max} - Machines(C_i=C_{max}) reproduction technique incorporates a second criterion. In this way, it considers that two solutions are equal if they have the same C_{max} and the same number of machines with a processing time equal to its makespan Machines(C_i=C_{max}). Similarly, the reproduction technique C_{max} - Average(C_i) uses two criteria. In this way, if two solutions have the same C_{max}, it calculates their average processing time of the machines Average(C_i). Thus, if the solutions have the same values of C_{max} and Average(C_i), they are considered repeated. Finally, the reproduction technique C_{max} - Machines(C_i=C_{max}) - Average(C_i) employees three criteria to determine that two solution are repeated. If two solutions have the same values of C_{max}, it verifies that they also have the same number of machines with Machines(C_i=C_{max}). Thus, if both criteria are met, it calculates the Average(C_i) of the solutions, and if they also match, they are considered repeated.

Table 8.5 contains the average RPD values obtained by the four reproduction techniques proposed in this phase. Each column indicates the results reached by each variant, identified by the criterion that they use to consider that two solutions are repeated as C_{max}, C_{max} - Machines(C_i=C_{max}), C_{max} - Average(C_i), and C_{max} - Machines(C_i=C_{max}) - Average(C_i). For a more detailed study, we compare the

Table 8.4: Comparison of the reproduction techniques: Proportional-Random, Proportional-Parents, and Proportional-Worst using RPD.

Instance Set		**Proportional Random**	**Proportional Parents**	**Proportional Worst**
n	100	0.034	0.034	**0.027**
	200	0.033	0.033	**0.026**
	500	0.033	0.033	**0.026**
	1000	0.035	0.035	**0.028**
m	10	0.034	0.033	**0.027**
	20	0.034	0.034	**0.027**
	30	0.034	0.034	**0.027**
	40	0.035	0.034	**0.027**
	50	0.035	0.034	**0.027**
p_{ij}	$U(1, 100)$	0.045	0.045	**0.042**
	$U(10, 100)$	0.038	0.038	**0.037**
	$U(100, 120)$	0.011	0.011	**0.010**
	$U(100, 200)$	0.026	0.026	**0.023**
	$U(1000, 1100)$	0.005	0.005	**0.004**
	$JobsCorre$	0.071	0.070	**0.039**
	$MacsCorre$	0.047	0.046	**0.040**
1400 instances		0.035	0.034	**0.028**

performance of the reproduction techniques using four different criteria to group the 1400 instances, (1) the number of machines (m), (2) the number of jobs (n), the distribution of the processing times (p_{ij}), and (4) the 1400 instances together.

From Table 8.5 can be concluded that the reproduction technique with the best performance is C_{max} - Machines(C_i=C_{max}) - Average(C_i) that uses three criteria to validate that two solutions are the same. These results also indicate that the reproduction technique with this upgrade improved the EGGA performance from an RPD value of 0.028 to 0.022, i.e., the criterion introduced to identify the repeated solutions provided an improved rate of about 27%. These results show the importance of using a reproduction technique that incorporates knowledge of the problem domain to efficiently handle the characteristics and conditions of the problem to solve.

8.3 Conclusions of the analysis

Although the crossover and mutation operators are crucial to the performance of a GGA, they cannot work well without an efficient reproduction technique. Given the importance of this GGA component, in this chapter, we studied how different strategies can impact the performance of a reproduction technique. To reach this goal, we

Table 8.5: Comparison of the strategies to sort the population: C_{max}, C_{max} - Machines(C_i=C_{max}), C_{max} - Average(C_i), and C_{max} - Machines(C_i=C_{max}) - Average(C_i) using RPD.

Instance Set		C_{max}	C_{max} **Machines**(C_i=C_{max})	C_{max} **Average**(C_i)	C_{max} **Machines**(C_i=C_{max}) **Average**(C_i)
n	100	0.035	0.030	0.023	**0.021**
	200	0.035	0.030	0.022	**0.021**
	500	0.035	0.031	0.022	**0.021**
	1000	0.036	0.031	0.024	**0.022**
m	10	0.035	0.030	0.023	**0.021**
	20	0.036	0.031	0.023	**0.021**
	30	0.036	0.031	0.023	**0.022**
	40	0.036	0.031	0.023	**0.022**
	50	0.036	0.031	0.023	**0.022**
p_{ij}	U(1, 100)	0.046	0.037	0.032	**0.028**
	U(10, 100)	0.037	0.031	0.025	**0.021**
	U(100, 120)	0.013	0.012	0.010	**0.009**
	U(100, 200)	0.029	0.027	**0.018**	**0.018**
	U(1000, 1100)	0.006	0.006	**0.003**	**0.003**
	$JobsCorre$	0.067	0.056	0.038	**0.037**
	$MacsCorre$	0.057	0.050	0.038	**0.037**
1400 instances		0.036	0.031	0.023	**0.022**

proposed an experimental study based on phases. In the first stage, we considered both random and biased strategies. The knowledge gained from this study was used to design a purpose-specific reproduction technique for $R||C_{max}$. The designed reproduction technique was incorporated into the EGGA with specific-purpose components designed with the systematical studies presented in previous chapters. The results suggested that the suitable reproduction technique for the EGGA that solves $R||C_{max}$ should use a strategy to arrange the solutions from best to worst, to later rearrange the population, placing at the end the solutions with the same makespan C_{max}, the same number of machines with $C_i = C_{max}$ and the same average processing time Average(C_i). Moreover, the reproduction technique should select the first individuals of the ordered population for crossover and introduce the offspring to the population, replacing the repeated solutions and later the solutions with the worst fitness. Finally, the selection and replacement strategies for the mutation process are kept. Therefore, the first solutions of the ordered population (from best to worst and with the repeated solutions placed at the end) must be mutated, and the cloned solutions must first replace the repeated solutions and then the solutions with the worst fitness. Hence, as the reproduction technique is the last GGA component studied, at this point of the research, we can make a comparison between the performance of the initial GGA and this latest version of the EGGA that includes the intelligent strategies designed based

on the knowledge obtained from the systematical analysis of the optimization process of each GGA component. In this way, EGGA includes the population initialization strategy: Random min (presented in Chapter 6), the crossover operator (introduced in Chapter 6), the mutation operators: 2-Items reinsertion (introduced in Chapter 7), and the reproduction technique designed using the knowledge generated from the study presented in this section, referred to as Ranking BRW (best, repeated, and worst). As the experimental results indicated, the systematical study of the GGA components in isolation paid off, improving from the RPD of the initial GGA of 0.069 to an RPD of 0.022 of the EGGA, which means an improvement rate of about 68%. Since the reproduction technique was the last GGA component to examine, in the next chapter we will conduct a characterization of the EGGA optimization process, to study the algorithmic behavior emerged when all the designed components work together.

Chapter 9

Study of the $R||C_{max}$ optimization process

This chapter presents an approach, based on exploratory data analysis techniques, to characterize the properties of $R||C_{max}$ instances and the algorithmic behavior of the EGGA that includes the intelligent strategies designed from the systematical studies presented in Chapters 5, 6, 7, and 8 for the population initialization strategy, the crossover operator, the mutation operator, and the reproduction technique, respectively. In this order of ideas, this section begins with a review of the state-of-the-art proposals for the characterization of combinatorial optimization problems (COPs). Later, it presents an experimental study of the $R||C_{max}$ properties and the EGGA optimization process that consists of (1) identifying the instance characteristics that give information useful to understand its structure and as well as the indexes to collect information related to the EGGA optimization process and its final performance, (2) refining the instance characteristics and the measures for the study of the EGGA optimization process to discard incorrect, redundant, or irrelevant indexes, (3) seeking for relations among the instance characteristics, the EGGA optimization process measures, and its final performance that explain the EGGA algorithmic behavior, and (4) understanding the EGGA algorithmic behavior, identify its opportunity niches, and improve its performance. The experimental results indicate that the difficulty of the $R||C_{max}$ instance are mainly related to its size and the distribution of its processing times. Furthermore, the study of relations among the $R||C_{max}$ and the optimization process suggested that the EGGA performance could be improved by incorporating a different strategy to generate the initial population as well as a heuristic strategy to provide more exploration. Lastly, this chapter presents the last improvement to the EGGA designed with the knowledge gained from this characterization study, referred to as the Final GGA (FGGA). Experimental results indicated that the improvements performed to the EGGA with the knowledge obtained from the characterization study allowed an improvement rate of about 63%. Therefore, this means an improvement rate of about 392% from the initial GGA to its final version.

9.1 Approaches for the characterization of COPs

One of the main challenges in the design of heuristic algorithms is to identify which strategies make an algorithm show better performance and under what conditions they obtain it. Much of the recent progress, in the development of algorithms, has been aided by a better understanding of the properties of problem instances and the performance of the algorithms that solve them. Examples of that are the numerous studies about the complexity of instances and the performance of algorithms for the Propositional Satisfiability Problem and the Traveling Salesman Problem [385, 386]. However, the percentage of studies carried out to understand how and why the algorithms follow particular behaviors is much lower than the works that create such algorithms. Furthermore, the characteristics that explain the degree of difficulty of the instances of many COPs have not yet been studied [387, 388].

The related works about the characterization and analysis of the algorithmic optimization process include different approaches, such as the algorithm selection problem introduced by Rice in 1976 [389]. The key idea of this proposal is to select an effective or best algorithm given some independent characteristics of the problems which are important for the algorithm selection and performance. From the seminal work of Rice, there have been important efforts to predict the performance of algorithms for COPs via the application of statistical methods, multivariate analysis, and machine learning techniques [390].

Another important field is meta-learning, an active area of research, especially concerning algorithm selection and configuration [391, 392]. Likewise, the analysis of the problem search space structure is another successful approach since it has allowed studying metrics to characterize the search space and its properties, used to analyze a wide variety of COPs [393, 394, 395].

On the other hand, studies of the relative hardness of instances for various NP-hard COPs have shown a phase transition property around which the most difficult problems occur. The identification of these properties has allowed characterizing the instances as easy or hard, and a large number of experimental results have demonstrated the effect of these properties for both exact and heuristics algorithms [396, 397].

The difficulty of the NP-hard COPs instances can also be characterized by studying the structure of optimal solutions. For some COPs, it has been possible to identify sets of backbone variables that have fixed values amongst all optimal solutions, showing that there is a relation between the magnitudes of these sets and the degree of the difficulty of the instances [398, 395].

Although various studies have focussed on measuring the instance difficulty for COPs, the complexities of NP-hard COPs and heuristic algorithms have shown that a single characterization approach is not sufficient to understand the performance of the algorithms and the difficulty of the instances. The state-of-the-art highlights the need for a more complete analysis, combining all the proposed characterization techniques to obtain better explanations regarding the performance of the heuristic algorithms and the difficulty of the instances solved. This type of study is important since it can provide a solid basis for the analysis and design of algorithms.

A good experimental analysis of heuristic algorithms should enable to understand the relationships between features of instances and algorithmic behavior, to explain the strengths and weaknesses of each algorithm. The definition of the optimal subset of features that adequately measure the relative difficulty of the instances is a critical task, the way instances structure affects the performance of the algorithms represent a big challenge and the explanations of the algorithmic behavior depend in an unstable way on the features actually used. The construction of features to characterize the structure of the instances of an optimization problem can be made via problem-independent metrics, considering the search space and its properties, and via problem-specific features. Smith-Miles and Lopes review some of the problem-specific features that have been constructed to characterize the problem difficulty of various COPs [399]. However, as far as we know, there is not any work related to the characterization of Parallel-machine scheduling problems. The motivation behind the work presented in this chapter is to contribute to the construction of suitable problem-specific features that allow us to characterize and understand the hardness of Parallel-machine scheduling problems.

9.2 Experimental study of the optimization process of $R||C_{max}$

This section describes the characterization process of the $R||C_{max}$ problem, the EGGA algorithmic behavior analysis, and its final performance. This study is conducted following the four phases of the approach proposed by Quiroz-Castellanos [400]. That is characterization, characteristics refining, relations study, and algorithmic behavior explanations.

The first stage, characterization, consists of identifying the most relevant properties of the instances of the problem studied, and measuring them via characterization functions (indexes), which provide valuable information that could enable us to explain the performance of the heuristics that solve them.

On the other hand, the characteristics refining phase involves the use of exploratory data analysis techniques to discard incorrect, redundant, or irrelevant indexes; if necessary, new indexes are incorporated, in order to characterize all the relevant factors that allow discriminating between instances with different features. Next, in the third phase, the relations study, an exploratory analysis of the final set of indexes is carried out to look for characteristics of the test instances explaining the algorithmic behavior of the heuristics under research. Finally, as the last phase of the approach, the knowledge resulting from this study conducts an understanding of the behavior of the heuristic algorithms, explaining how their final performance is affected by several factors that cause the features of the instances.

The following sections present the results obtained of applying the characterization method to the EGGA, created based on the intelligent strategies designed in Chapters 4, 5, 6, 7, and 8.

9.2.1 Phase 1: Characterization

The main objective of the characterization phase is to identify and measure relevant factors that describe the structure of the instances, the algorithmic behavior, and the final performance of the algorithm studied. In this section, we describe the set of indexes used to measure the most relevant properties of the 1400 test instances of the $R||C_{max}$ problem, the EGGA behavior, and its final performance.

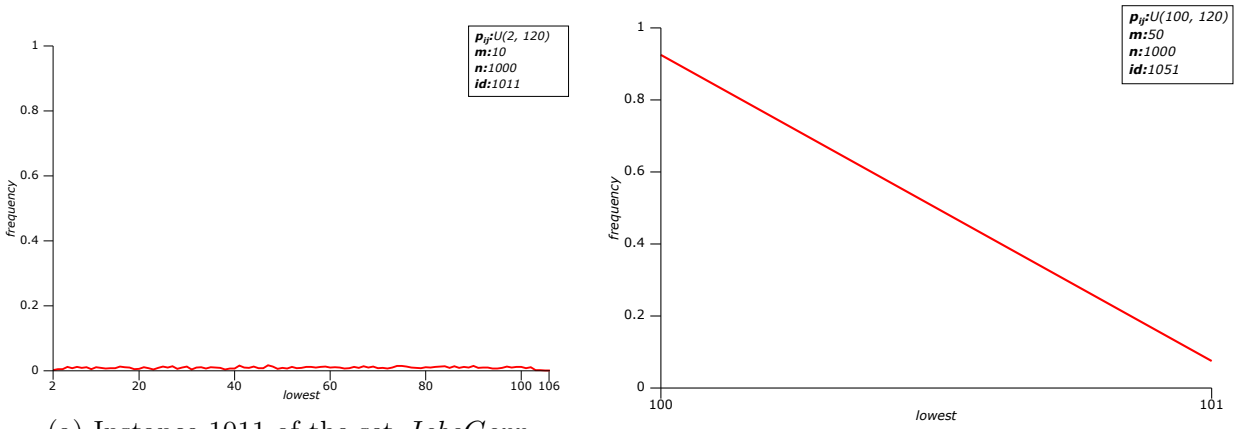
Measuring $R||C_{max}$ instances hardness

Knowing and understanding the structure of the $R||C_{max}$ instances play an important role in predicting the quality of the solutions generated by the metaheuristic algorithms that address them. It is well-known that factors such as the number of jobs and machines, the central tendency of their processing times, and their distribution, influence on the difficulty level that an instance may have for a solution algorithm. However, the challenge is the formulation of indexes able to characterize these factors. Like this is the first effort to characterize $R||C_{max}$, we apply several general-purpose indexes, based on descriptive statistics, to analyze different characteristics of the problem and collect relevant information about the parameters of every instance. The detail of the instances used in this work can be found in Section 3.2.

The characteristics used to generate this benchmark are the first that allow us to discriminate between instances with different structures. In this sense, given an instance of $R||C_{max}$, we use the number of machines m and the number of jobs n to measure the size of the problem. Another important factor to characterize $R||C_{max}$ is the distribution of the processing times p_{ij}, which can be studied measuring its form, centralization, dispersion, as well as the correlation concerning the jobs and the machines.

Recalling from Section 3.1, constructive heuristics for the Parallel-machine scheduling problem consider different properties to classify the jobs before scheduling them (see Figure 5.1), like the processing time p_{ij} required by the fastest machine to process each job (*lowest*) and the difference between the two fastest machines to process each job (*diff_fastest*). In order to explore the impact of these properties on the performance of the algorithms, we generate plots for each instance to observe the way these characteristics can vary among different instances. For example, the *multiplicity* of the instance property *lowest*, represented as a vector that contains the processing times p_{ij} required by the fastest machine i to process each job j, is an aspect that could incise on the difficulty of the instances. Figure 9.1 holds information about it, which allows us to analyze how the *multiplicity*(*lowest*) varies from one instance to another. To generate this figure, first, for each instance I, the set *lowest* was created with the minimum processing time p_{ij} required to process every job j, to later identify the different values in *lowest*, and count its frequency. In this way, Figure 9.1 contains the instances of the extremes, i.e., the instance where the most frequent processing time is the highest and the one with the lowest of the 1400 cases. In each graph, the $x-$axis depicts the processing time of the jobs, and the $y-$axis indicates its frequency as a percentage of the total number of jobs. Besides, in the upper right corner, each figure

holds the general characteristics of the plotted instance, including its id, the range used to generate its values of p_{ij}, the number of jobs n, and the number of machines m. As can be seen in these graphs, the benchmark considered in this work contains instances where the lowest processing times are not very repeated, such as the one shown in Figure 9.1a with 0.16 percent incidence. In contrast, Figure 9.1b indicates that this benchmark also includes instances with the lowest processing times repeated up to 92.5 percent.



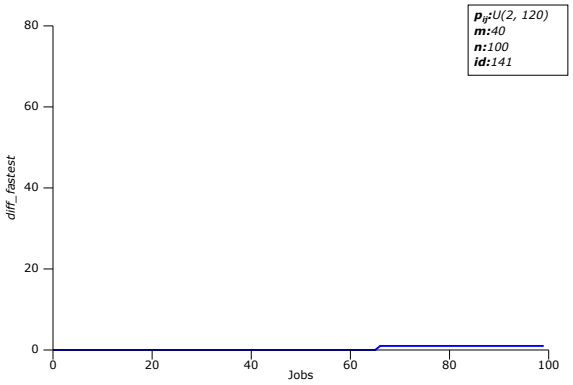
(a) Instance 1011 of the set $JobsCorr$

(b) Instance 1051 of the set $U(100, 120)$

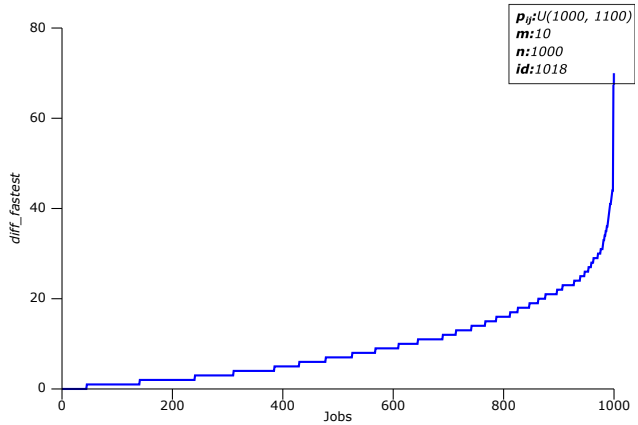
Figure 9.1: $Multiplicity$ of the lowest processing times of jobs ($lowest$).

Another aspect that could be related to the difficulty of the instances is the difference between the processing times of the two fastest machines to process each job ($diff\_fastest$), mainly for metaheuristic algorithms that allocate the jobs using this information (see Figure 5.1). Figure 9.2 shows how the values of this feature vary from one instance to another. To generate this figure, for each instance I, the set $diff\_fastest$ is created, to later sort the jobs in increasing order according to this property. In this way, Figure 9.2 holds the plots for the instances with the greatest and the smallest value in $diff\_fastest$ of the 1400 cases, respectively. In each graph, the $x-$axis depicts the jobs in increasing order according to their values in $diff\_fastest$, and the $y-$axis indicates the difference between the processing times of the two fastest machines to process each job. Besides, the upper right corner of each graph presents the general characteristics of the plotted instance: the criterion used to generate its values of p_{ij}, the number of machines m, the number of jobs n, and the identifier id. From these figures can be observed that, in some instances, the difference between the two fastest machines to process each job is small, like the one plotted in Figure 9.2a with differences of 0 and 1 only. In contrast, Figure 9.2b shows that the benchmark also includes instances with values of $diff\_fastest$ that can vary between 0 and 70. This type of information can be helpful to build a solution since it indicates that, in some cases, the fastest machines to process all jobs have similar speeds. Consequently, placing the jobs in the fastest or the second-fastest machine affects less than when there exists a wide variation among the speed of the two fastest machines.

Table 9.1 contains the general indexes proposed to analyze the 1400 instances, where the first column includes the name of each index, and the second one its description.



(a) Instance 141 of the set $JobsCorr$



(b) Instance 1018 of the set $U(1000, 1100)$

Figure 9.2: Difference between the processing times of the two fastest machines to process each job (*diff_fastest*).

In this sense, for each instance I, we collect the number of jobs n and machines m; the quotient of the number of jobs by the number of machines $\frac{n}{m}$; the minimum $min(p_{ij})$ and the maximum $min(p_{ij})$ processing time; the average $mean(p_{ij})$ and the coefficient of variation of the processing times $cv(p_{ij})$; the difference $range(p_{ij})$ and the quotient (q) between the maximum and minimum processing times; and the level of correlation among the processing times of jobs (job_corre) and machines ($mach\_corre$). These indexes allow measuring the general characteristics of the structure of the instances. Equation 9.1 shows how to calculate the job correlation of an instance I, where $range(p_j)$ indicates the subtraction of the largest and minor processing time of the job j, n represent the number of jobs, and $range(p_{ij})$ depicts the difference between the highest and the lowest processing time of I, respectively. In this way, as the value of job_corre of I approaches 1, its jobs are more correlated, indicating that its jobs have similar processing times for all machines (either short, medium, or long). Similarly, Equation 9.2 is used to measure the machine correlation ($mach\_corre$) of an instance I, where $range(p_i)$ collects the subtraction of the highest and the lowest processing time of the machine i, m indicates the number of machines, and $range(p_{ij})$ depicts the difference between the largest and minor processing time of I, respectively. Like job_corre, as the value of $mach\_corre$ of I approaches 1, its machines are more correlated, indicating that its machines have similar speeds to process all the jobs (either fast or slow).

$$job\_corre = 1 - \frac{\frac{\sum_{j=1}^{n} range(p_j)}{n}}{range(p_{ij})} \tag{9.1}$$

$$mach\_corre = 1 - \frac{\frac{\sum_{i=1}^{m} range(p_i)}{m}}{range(p_{ij})}, \tag{9.2}$$

Table 9.1: General characteristics for I.

Index	**Description**
n	Number of jobs to be scheduled.
m	Number of machines available.
$\frac{n}{m}$	Quotient of n between m.
$min(p_{ij})$	Minimum processing time in I.
$max(p_{ij})$	Maximum processing time in I.
$mean(p_{ij})$	Average processing time in I.
$cv(p_{ij})$	Coefficient of the processing times variations in I.
$range(p_{ij})$	Difference between $G_{min}(I)$ and $G_{max}(I)$ in I.
q	Quotient of $max(p_{ij})$ between $min(p_{ij})$.
job_corre	Level of correlation among jobs in I.
$mach\_corre$	Level of correlation among machines in I

Additionally, different features of jobs and machines are independently analyzed using the seven descriptive measures defined in Table 9.2. In this table, the first column includes the name of each index, and the second one holds its description. As can be seen from this table, each index requires an argument (X) that can represent either a set of processing times of a job, a machine, or any specific feature extracted from a given instance I. For example, the m processing times p_{ij} from p_{1j} to p_{mj} of each job j are used to create the set X to analyze the general structure of the jobs. In this way, the seven measures listed in Table 9.2 (i.e., $mean$, cv, $uniformity$, min, max, $range$, and $multiplicity$) are applied to each job independently, to later calculate the average of each measure regarding the n jobs under study.

The information collected with these measures and the process mentioned above allow examining the general distribution, uniformity, form, and size of the jobs in a given instance. Furthermore, these measures can also be used to study more specific characteristics of jobs, like the behavior followed by the shortest, the longest, and the average processing time of jobs. To collect this information, first, the characteristic of interest is extracted from each job and then used as the argument (X) of the measures to be calculated. Thereby, data related to the characteristic of interest, like its central tendency, dispersion, and uniformity, can be explored using the indexes listed in Table 9.2.

Similar to jobs, several machine characteristics are studied using the seven descriptive measures detailed above. In this sense, to analyze the general structure of machines, the measures are applied to each machine independently. That is, forming the set X with the n possible processing times p_{ij} for each machine i, from p_{i1} to p_{in}. Furthermore, more specific features of machines are explored, like the time required by each machine to process the job with the shortest processing time ($shortest$), the number of jobs that each machine processes faster than the other ones ($n\_fastest$), the average speed of machines considering the n jobs ($average\_speed$), and the average speed of machines considering only the jobs that each machine process faster than the others ($average\_speed\_fastest$).

Table 9.2: Descriptive measures.

Index	**Description**
$mean(X)$	Mean of the data set X.
$cv(X)$	Coefficient of variation of the data set X.
$uniformity(X)$	Uniformity of the data set X [400].
$min(X)$	Minimum value in the data set X.
$max(X)$	Maximum value in the data set X.
$range(X)$	Difference between the maximum and the minimum value in X.
$multiplicity(X)$	Multiplicity of the data set X [400].

In addition to this, we used indexes with specific-purpose. For example, to analyze relations jobs-machines we use four indexes: *above* $\frac{n}{m}$, *equal* $\frac{n}{m}$, and *below* $\frac{n}{m}$. To calculate these indexes, first, the vector $n\_fastest$ with length m is created where each position of the vector includes the number of jobs that each machine processes faster than the other ones. In this way, the characteristics of this property can be studied, applying the indexes of Table 9.2. Additionally, we measure *above* $\frac{n}{m}$ that counts the number of machines processing faster a lower number of jobs that $\frac{n}{m}$, *equal* $\frac{n}{m}$ that indicates the number of machines processing faster an equal number of jobs that $\frac{n}{m}$, and *below* $\frac{n}{m}$ that measures the number of machines processing faster a lower number of jobs that $\frac{n}{m}$. However, it is important to note that these indexes are not standardized.

On the other hand, we also use the descriptive measures in Table 9.2 to collect information related to the difference between the processing times of the two fastest machines to process each job (see Figures 5.1 and 9.2). The information collected from this property is important since it allows detecting the contrast among processing the jobs by the machine that processes them fastest or the second one. Recalling from Figure 9.2b, this difference can be considerably large. To generate this data, first, the vector *diff_fastest* is created with the difference between the two fastest machines to process each job, to later use the seven indexes listed in Table 9.2 to analyze its central tendency, uniformity, form, and multiplicity.

Finally, we use the indexes in Table 9.2 to analyze data of the instances related to the instance property *lowest* (see Figure 5.1 and 9.1). The study of this property (*lowest*) is very useful since it provides information related to the shortest time in which the jobs of an instance I can be processed. In this way, indexes of Table 9.2 bring information with respect to its central tendency, distribution, form, and multiplicity. Additionally, we apply the indexes max_repe, introduced by Quiroz-Castellanos in [400] as well as $no\_multiplicity$, introduced in this work. Max_repe indicates the frequency of the most recurrent processing time, and $no\_multiplicity$ counts the number of processing times that only appear one time. Likewise, it is important to take into account the characteristic *lowest*, since one of the first ideas that can arise at the moment of solving an $R||C_{max}$ problem is to assign the jobs to the machine that processes them faster. In this sense, the information collected from this property can help to validate the use of said heuristic or discard it and guide the selection of a new heuristic.

Measuring the EGGA Behavior

This section describes the indexes used to measure the algorithmic performance of the EGGA, proposed based on the studies conducted in Chapters 4, 5, 6, 7, and 8. The main objective of this section is to understand the operational functioning of the EGGA, in order to identify possible niches of opportunity to improve its performance. In this sense, we applied different measures to analyze the efficiency and usefulness of each EGGA component, including the strategy to generate the initial population, the crossover and mutation operators, as well as the selection and replacement mechanisms. The studied indexes seek to characterize the individual capabilities of each operator, as well as their contribution to the performance of the EGGA by working all together.

To analyze the algorithmic behavior of each EGGA component, we use the seven descriptive measures described in Table 9.2 together with an extra index to measure the number of repeated solutions $repe(X)$ that receives a set X of solutions and counts how many of them share the same C_{max}, the number of machines with $C_i = C_{max}$, and the average processing time of the machines $average\ (C_i)$. It is important to note that although these three criteria do not guarantee that two solutions are selfsame, they help to identify very similar solutions to measure the population convergence degree.

In this way, we use the aforementioned indexes to analyze the central tendency, dispersion, and uniformity of the population at different stages of the search. For example, to study the algorithmic behavior of the population initialization strategy, we apply the indexes to the initial population. That is, we create the vector X with the C_{max} of each solution to recognize the characteristics of the initial solutions generated by the population initialization strategy, in order to identify its strengths and weaknesses. Similarly, we apply the indexes to the final solution set to analyze the search abilities that the EGGA components have when working together. Additionally, we employ these indexes to analyze different aspects of the population generated in each generation. To conduct this study, we generate eight vectors, one for each descriptive measure: $min()$, $max()$, $distance$, etc., and one for the repeated solutions $repe$. Each vector collects the result of applying a descriptive measure to the population of every generation. Therefore, they were called $mins$, $maxs$, $distances$, etc. When the search process ends, the seven indexes are applied again to each of those vectors. Thus, we can study in detail the exploration and exploitation capabilities of the EGGA to try to identify whether it stagnated in local optima or it maintained diversity in population throughout the search. That is, we generate $8 \times 7 = 56$ indexes for this purpose.

In addition, we measure other indexes such as the success rate of the crossover $success\_rate(crossover)$ and mutation $success\_rate(mutation)$ operators to identify the percentage of solutions that improve their quality when going through these processes. We consider that the crossover process was successful if the solution generated is better than at least one of the two parents. In the case of mutation, the operation is considered successful if the generated solution improves the quality of the initial solution. Finally, the *solver* index stores the strategy that generates the best solution during the search, which allows identifying the individual contribution of each component to the final performance of the EGGA. Thus, *solver* stores 0 if the EGGA does not find a solution equal to or better than CPLEX. Otherwise, it saves 1 if the

best solution is found by the population initialization strategy, 2 if it is created by the crossover operator, and 3 if it is generated by the mutation operator.

The fifty-nine indexes described above were used to collect information on the performance of the EGGA. As can be seen, the number of indexes studied is somewhat large, so in the next stage (refining) the indexes that provide the most useful information will be identified to avoid redundancy and future problems during the analyses stage.

Measuring the EGGA Final Performance

One of the most important aspects to consider during the characterization of the EGGA behavior is its final performance. Therefore, this section presents the indexes used to collect this information. In this way, we cover the three elements necessary to solve a combinatorial problem like $R||C_{max}$, i.e., the input (instance characteristics), the process (algorithmic behavior), and the output (final performance).

A significant index in the algorithmic behavior analyses the quality of the solutions obtained. To characterize the EGGA final performance, we use the Relative Percentage Deviation (RPD). A value $RPD = 0$ means that the C_{max} found is equal to that found by two hours of CPLEX, a negative value indicates that a better solution is found by EGGA, and a positive value means that the result of EGGA is worse than the CPLEX result. Another important index is the time factor, we characterize it using two indexes: the real-time (t) that the algorithm required to try to solve the problem, and the number of generations g that the algorithm used to find the best solution. As can be seen, the indexes used to measure the final performance of EGGA are few; therefore, no refinement will be necessary.

9.2.2 Phase 2: Characteristics Refining

Once $R||C_{max}$ interest indexes have been defined, it is necessary to perform an exploratory analysis of the variables of interest to eliminate redundant measures, discard incorrect and irrelevant indexes and validate the proposed indexes. During this stage, first, we analyze the relationships among each pair of indexes used to characterize the $R||C_{max}$ problem, looking for redundant, irrelevant, and incorrect indexes. In this manner, we generated and analyzed a correlation matrix with those variables. As a result, we observed that some pairs of indexes measuring features related to the processing times of the $R||C_{max}$ instances collect redundant information since they are strongly correlated. Finally, the conducted study enabled observing the contribution and consistency of each index, detecting those that do not hold meaningful information for the characterization of the structure of the $R||C_{max}$ instances. Thus, we discarded inconsistent, irrelevant, and redundant indexes to avoid the noise provided by them to the analysis of the instance structure. Table 9.3 includes the detail of the final set of indexes used to analyze the structure of the $R||C_{max}$ instances. The first column indicates the type of information that collects, while the second one holds the name of every index.

Table 9.3: Final set of indexes for $R||C_{max}$ characterization.

Type	**Index**
Size	n m q
Centralization	$mean(p_{ij})$ $mean(diff\_fastest)$
Dispersion	$range(p_{ij})$ $cv(p_{ij})$ $cv(min(p_j))$ $cv(n\_fastest)$
Form	$uniformity(p_i)$ $uniformity(p_j)$ $uniformity(n\_fastest)$
Location	$max(p_{ij})$ $min(p_{ij})$
Relations job-machine	$\frac{n}{m}$ $below\ \frac{n}{m}$
Machine	$max(diff\_fastest)$ $mach\_corre$
Jobs	$multiplicity(lowest)$ job_corre

After identifying the final set of indexes to characterize the structure of the $R||C_{max}$ test instances, we conduct a Principal Components Analysis (PCA), that allows plotting eight subsets of instances, arranged according to Table 9.4. In this table, the first column indicates the processing time distribution of each collection of instances, followed by its number of jobs, its number of machines, and its identifier from 1 to 8. Figure 9.3 contains the instances of the eight groups plotted according to the first three principal components. As can be seen in this figure, the instances of each group are together, and the groups do not overlap. That is, the first three components are enough for discriminating among instances with different characteristics. The first component, called relation job-machine, comprises indexes like the number of machines m and the quotient of the number of jobs by the number of machines ($\frac{n}{m}$) associated with the size of the problem. Furthermore, it contemplates the form of the processing time distribution of the jobs (i.e., the uniformity of the jobs $uniformity(p_j)$) and variables that collect information from the machines that process fastest the jobs using the *diff_fastest* vector that includes the difference between the two fastest machines to process each job. On the other hand, the second component, called job structure, contemplates characteristics of the instances related to the form (uniformity) and dispersion (range and coefficient of variation) of the processing time of the jobs. Finally, the third component, machine structure, embraces information related to the form of the processing time distribution of the machines (i.e., uniformity of the machines $uniformity(p_i)$), the number of jobs that each machine could process in an extreme

case n, and the frequency between the processing times required by the fastest machine to processes each job $multiplicity(lowest)$.

Table 9.4: Characteristics of the eight groups of instance. p_{ij}: processing time distribution. n: number of jobs. m: number of machines. $identifier$: identifier of each collection of instances.

p_{ij}	n	m	id
$U(1,\ 100)$	100	10	1
		50	2
	1000	10	3
		50	4
$U(1000,\ 1100)$	100	10	5
		50	6
	1000	10	7
		50	8

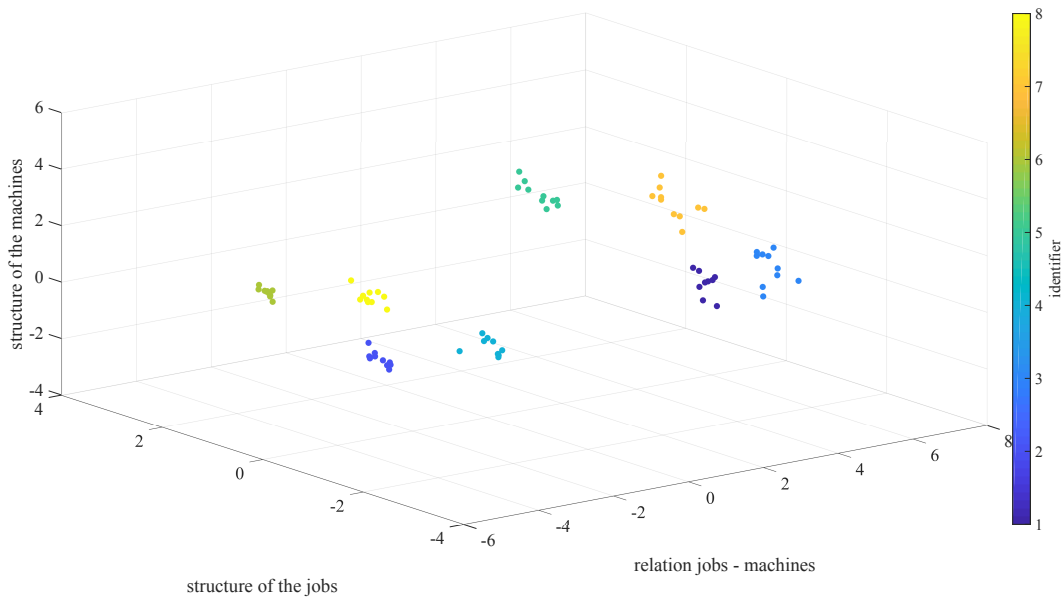


Figure 9.3: PCA of the twenty characteristics studied.

Once identified the indexes to characterize the problem structure, we proceed to refine the indexes proposed to measure the algorithmic behavior as follows. First, we analyze the relationships among each pair of the fifty-nine measures used to characterize the EGGA behavior, looking for the ones that collect the most useful information to understand the EGGA algorithmic behavior, and discarding the redundant or irrelevant ones. The selected indexes were compared versus the final set of the problem structure indexes and the final performance measures. Thus, we selected the algorithmic behavior measures presenting the higher correlations. The next section includes the correlation matrices with the indexes and measures that provide relevant information about the EGGA algorithmic behavior, together with 3D scatter plots that explain these relations.

9.2.3 Phase 3: Study of relations

The main goal of this phase is to explain the EGGA algorithmic behavior under research. Hence, we use different data analysis techniques to try to understand how the properties immersed in the $R||C_{max}$ instances influence on its difficulty. In this fashion, we use the information collected by the indexes and the measures to look for relations between the structure of the $R||C_{max}$ instances, the EGGA behavior during the search process, and its final performance.

The lineal association analysis was the first technique used to look for immersed relations among the three types of indexes: problem structure (input), algorithmic behavior (process), and final performance (output). This study was conducted as follows.

First, we identify the problem characteristics strongly related to the final performance of the EGGA (RPD and g), where we observe that the indexes with the highest correlation are the number of jobs n and the variation coefficient of the minimum processing times of the jobs $cv(min(p_j))$. Subsequently, these indexes were compared against the measures of algorithmic behavior to try to find input, process, and output relationships that allow understanding the EGGA optimization process. Below are the correlation tables and 3D scatter plots that help to understand the algorithmic behavior of the different EGGA components.

In order to individually analyze the algorithmic behavior of the main EGGA components and their contribution to the search process, in this stage, we identify the characteristics of the problem and the measures of the final performance related to the optimization process of each component. One of the most important EGGA components is the strategy to generate the initial population. The aspects to consider during the design of this strategy depend on the characteristics of the problem to solve. Therefore, it is relevant to know the performance and contribution of this component to the EGGA final performance. Table 9.5 shows the correlation matrix of the two measures that provide more information to understand the algorithmic behavior of the population initialization strategy. $average(ini\_pop)$: saves the mean quality of the solutions, which, being normalized based on the best solution found by CPLEX, helps to see the distance of the population to CPLEX. On the other hand, $cv(ini\_pop)$ measures the diversity of the population based on the quality of the solutions.

Table 9.5 shows the correlation matrix with the selected indexes to analyze the behavior of the strategy to initialize the population. We use two problem characteristics: n to measure the number of jobs and $cv(min(p_j))$ to collect the coefficient of variation of the minimum processing times of the jobs; two final performance measures: RPD that indicates the average percentage deviation from the solutions found by EGGA to CPLEX and g that indicates the generation in which EGGA found the best solution; and two measures to analyze the algorithmic behavior of the population initialization strategy: $average(ini\_pop)$ that indicates the average RPD from initial population to CPLEX and $cv(ini\_pop)$ that collects information related to the diversity of the initial solutions.

The correlation matrix indicates that n has a negative correlation with the algorithmic

Table 9.5: Problem characteristics, final performance measures, and indexes to analyze the algorithmic behavior of the population initialization strategy.

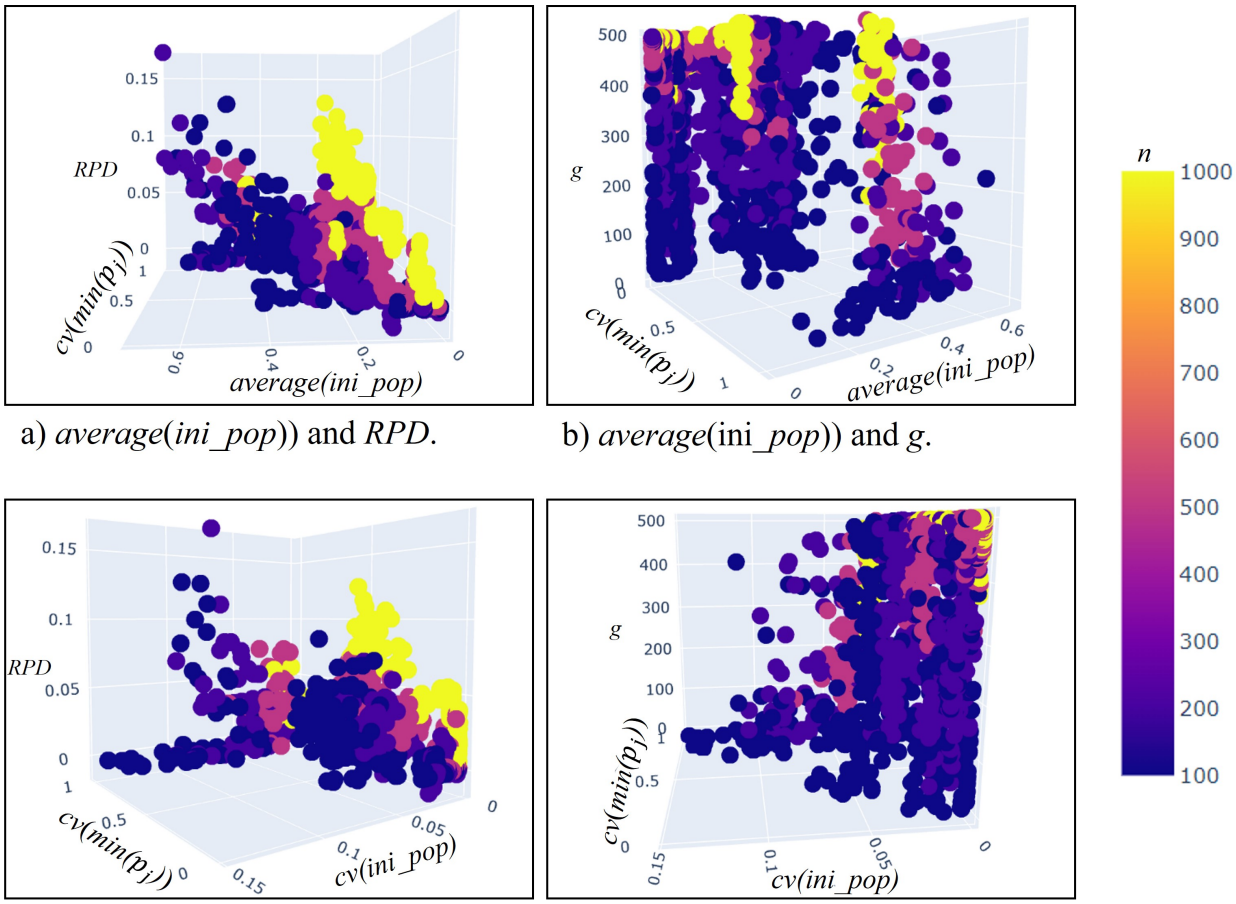
	n	$cv(min(p_j))$	$average(ini\_pop)$	$cv(ini\_pop)$	RPD	g
n	1.00					
$cv(min(p_j))$	0.0052	1.00				
$average(ini\_pop)$	-0.1821	0.7276	1.00			
$cv(ini\_pop)$	-0.4351	0.5541	0.7916	1.00		
RPD	0.4167	0.3800	0.2853	0.0590	1.00	
g	0.6270	-0.1502	0.3610	-0.6028	0.1867	1.00

behavior measures $average(ini\_pop)$ and $cv(ini\_pop)$. That is, the greater the number of jobs (n) in the instance to solve, the initial population strategy generates the solutions closer to the best-known (small $average(ini\_pop)$ values), and at the same time, the initial population has less diversity (low $cv(ini\_pop)$ values). In contrast, n has a positive correlation with the final performance measures RPD and g. Thus, as the number of jobs n increases, the distance between the solutions found by EGGA and CPLEX becomes larger (high RPD values), and the generation in which the best solution is found also grows (high g values). On the other hand, $cv(min(p_j))$ has a positive correlation with three measures studied. In this way, when the $cv(min(p_j))$ grow (the minimum time to process the jobs are more diverse), the initial population moves away from the best-known solution (high $average(ini\_pop)$ values) and the initial population has greater diversity (high $cv(ini\_pop)$ values). Regarding the final performance, none of the measures has a strong relationship with $cv(ini\_pop)$. However, the correlation matrix indicates that, when the initial population is generated far from the best-known solution (high $cv(ini\_pop)$ values), the quality of the best solution found by EGGA decreases (high RPD values), and the search gets stuck at local optima (low g values).

Figure 9.4 contains four 3D scatter plots with the problem characteristics n and $cv(min(p_j))$ and the final performance measures RPD and g that give more information about the performance of the strategy to generate the initial population. The four graphs include the two indexes n and $cv(min(p_j))$ and can vary in performance measures and measures to analyze the behavior of the initialization strategy $average(ini\_pop)$ and $cv(ini\_pop)$, which indicate the average distance of the initial solutions to the best-known solution and the diversity of the population, respectively.

The following list enumerates the conclusions obtained during the analysis of the correlation matrix that can be graphically observed in the four 3D scatter plots (Figure 9.4).

- Figure 9.4a shows that the EGGA finds better solutions (small RPD values) when the initial population is closer to the best-known solution (small $average(ini\_pop)$ values).
- Figure 9.4b indicates that EGGA sparingly uses generations (high g values) when the initial population is closer to the best-known solution (small $average(ini\_pop)$ values).



a) *average*(*ini_pop*)) and *RPD*.

b) *average*(ini_*pop*)) and *g*.

c) *cv*(ini_*pop*)) and *RPD*.

d) *cv*(ini_*pop*)) and *g*.

Figure 9.4: Scatter plots for the population initialization strategy.

- Figure 9.4c shows that the EGGA locates better solutions (small RPD values) when the initial solutions are similar (small $cv(ini\_pop)$ values).
- Figure 9.4d indicates that the EGGA makes better use of generations (high g values) when the initial solutions are similar (small $cv(ini\_pop)$ values).
- Figures 9.4a and 9.4c allow observing that, for each number of jobs n, EGGA finds better solutions (small RPD values) in instances with less variety in the minimum processing times of the jobs $cv(min\ (p_j)$.
- Figures 9.4b and 9.4d show that EGGA makes better use of generations (high g values) in instances with high $cv(min(p_j)$ values.

Another important aspect related to the EGGA performance is diversity, which has an important role to avoid a local optima stagnation. Table 9.6 shows the correlation matrix with the measures that provided the most useful information, selected following the same process, about the way EGGA handles the population diversity during the search process. $cv(ini\_pop)$ saves the diversity of the initial populations, $cv(fin\_pop)$ indicates the variance of the final population, $average(cvs)$ collects the average diversity

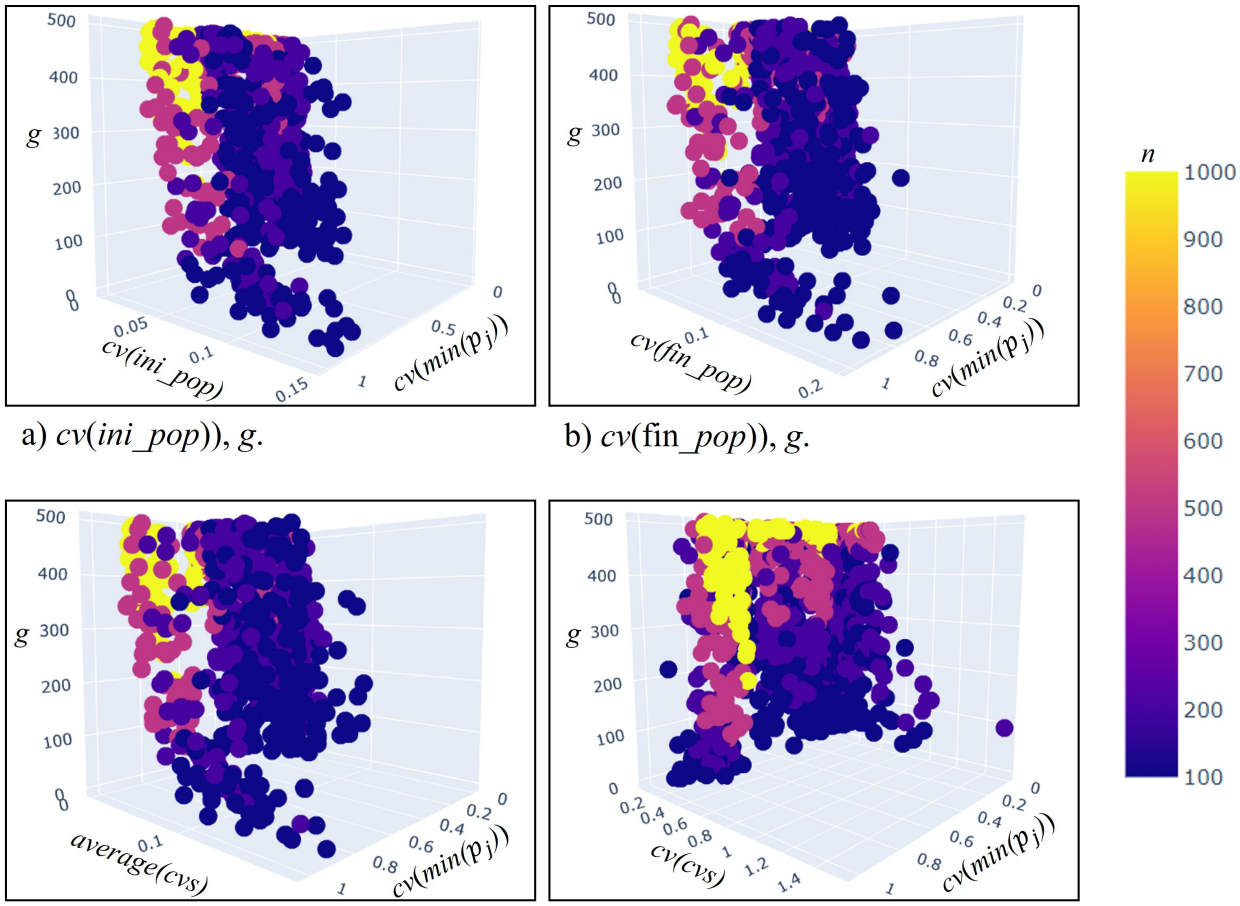
Table 9.6: Problem characteristics, final performance measures, and, indexes to analyze the way EGGA handles diversity.

	n	$cv(min(p_j))$	$cv(ini\_pop)$	$cv(fin\_pop)$	$average(cvs)$	$cv(cvs)$	g
n	1						
$cv(min(p_j))$	0.0052	1					
$cv(ini\_pop)$	-0.4350	0.5540	1				
$cv(fin\_pop)$	-0.4560	0.5218	0.8199	1			
$average(cvs)$	-0.5017	0.4840	0.8548	0.9607	1		
$cv(cvs)$	-0.0497	-0.1503	0.0264	-0.1602	-0.1338	1	
g	0.6268	-0.1502	-0.6028	-0.4682	-0.5097	-0.2479	1

of the entire search process, and $cv(cvs)$ indicates the way the variance keeps or changes in all the generations. In this study, we use the final performance measure g, which saves the generation in which the best solution is found. In this way, the measures and indexes highly correlated indicate how the diversity in the population influences the EGGA stagnation in local optima. Table 9.6 suggests that when EGGA tries to solve instances with many jobs (high values of n), the population of solutions starts and ends with little diversity (low values of $cv(ini\_pop)$ and $cv(end\_pop)$). On the contrary, the variance is greater in instances where the minimum processing times of the jobs vary (high values of $cv(min(p_j))$). Also, when the initial population has a lot of variance (high values of $cv(ini\_pop)$), EGGA gets stall easily in local optima (low values of g). This effect is more marked in instances with high coefficients of variation in the minimum processing times of the jobs (high values of $cv(min(p_j))$). Similarly, Table 9.6 suggests that EGGA gets stall when it promotes too much diversity throughout the search process (low values of g for high values of $average(cvs)$, $cv(cvs)$ and $cv(fin\_pop)$).

In order to graphically see the way EGGA handles diversity during the search process, Figure 9.5 includes four 3D scatter plots with the problem characteristics n and $cv(min(p_j))$ and the final performance measure g. Each graph differs in the measure used to analyze the diversity. The four graphs in Figs 9.5a-9.5d reveal that EGGA stagnates faster when the diversity is high in the initial population $cv(ini\_pop)$, during the search process $average(cvs)$ and $cv(cvs)$, and in the final population $cv(fin\_pop)$.

Like diversity, convergence is another important aspect to take into account to analyze the EGGA performance. Since it, together with diversity, brings EGGA the exploration-exploitation capability. Therefore, it is important to keep a balance between diversity and convergence during the entire search process. Table 9.7 presents the correlation matrix with the measures that provided the most useful information about the way EGGA handles the population convergence during the search process. $average(ini\_pop)$ saves the convergence of the initial population, $average(fin\_pop)$ indicates the convergence of the final population, $average(averages)$ collects the average convergence of the entire search process, and $cv(averages)$ indicates the way the convergence keeps or changes during the search process. In this study, we use the final performance measure RPD, which indicates the distance from the best solution found by EGGA and CPLEX. Hence, the highly correlated measures and indexes indicate how the EGGA convergence capability influences its final performance. Table 9.7



a) $cv(ini\_pop))$, g.

b) $cv(\text{fin}\_pop))$, g.

c) $average(cvs))$, g.

d) $cv(cvs)$, g.

Figure 9.5: Scatter plots to analyze the way EGGA handles the diversity.

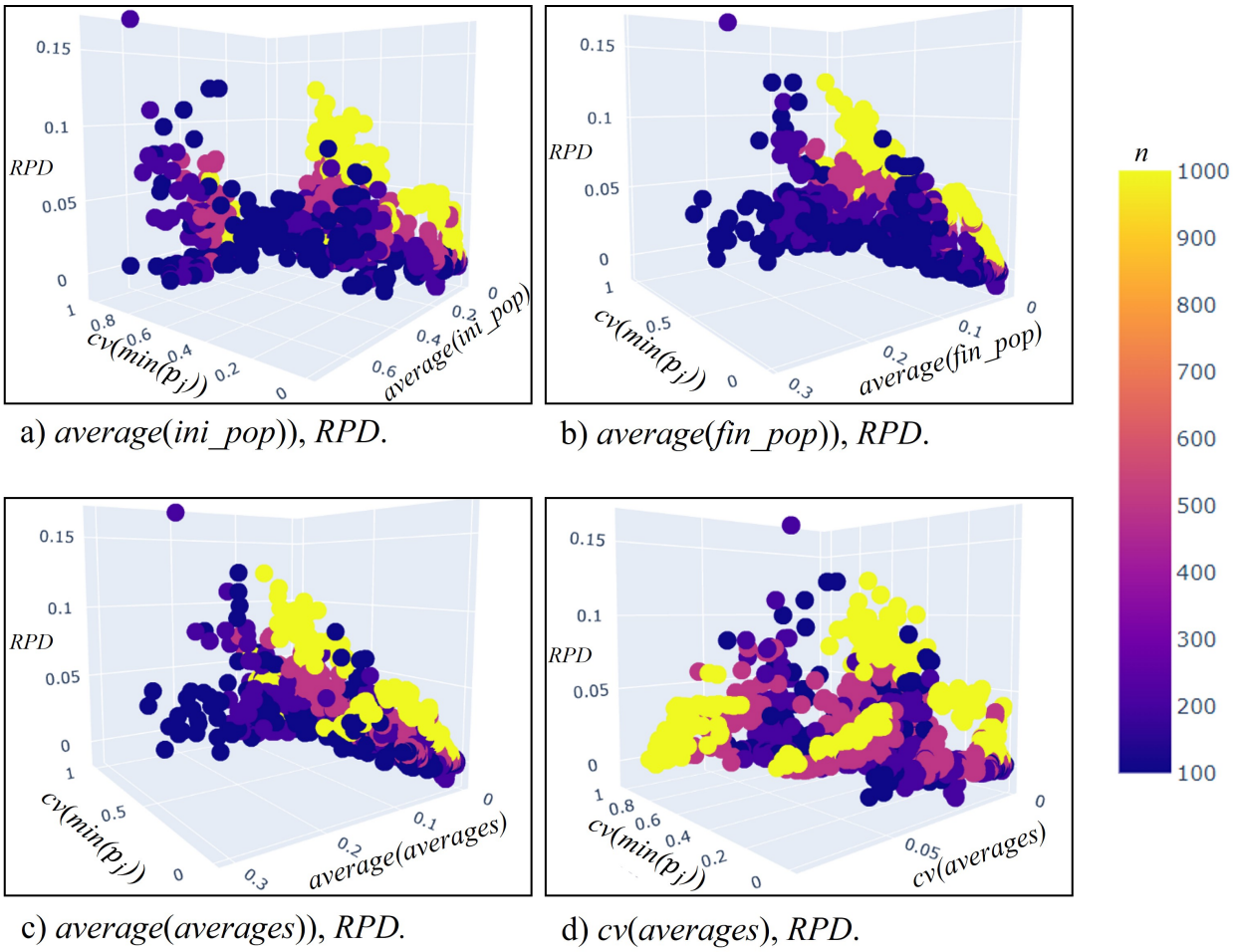
suggests that the most difficult instances are those with more jobs (high n values) and those with the highest coefficient of variation in the minimum processing time of the jobs (high $cv(min(p_j))$ values. Likewise, Table 9.7 indicates that as an instance has a higher $cv(min(p_j))$ value, the population of solutions starts and ends further from the best-known solution. Also, when the initial population is far away from the best-known solution (high values of $average(ini\_pop)$), EGGA does not have high-quality solutions (high RPD values). The same phenomenon occurs when the population stays away from the best-known solution throughout the search $average(averages)$. Accordingly, if the final population ends far from the best-known (high values of $average(fin\_pop)$), the EGGA performance is not good.

To graphically analyze the way EGGA convergence capability during the search process, Figure 9.6 presents four 3D scatter plots with the problem characteristics n and $cv(min(p_j))$ and the final performance measure RPD. Each graph differs in the measure used to analyze the convergence. The four graphs in Figs 9.6a-9.6d suggest the quality of the solutions found by EGGA decreases when the initial population is generated far from the best-known solution, the population keeps far from the well-known solution

Table 9.7: Problem characteristics, final performance measures, and indexes to analyze the way EGGA handles the convergence.

	n	$cv(min(p_j))$	$average(ini\_pop)$	$average(fin\_pop)$	$average(averages)$	$cv(averages)$	$average(cvs)$	RPD
n	1							
$cv(min(p_j))$	0.0052	1						
$average(ini\_pop)$	-0.1821	0.7276	1					
$average(fin\_pop)$	-0.0850	0.5542	0.5168	1				
$average(averages)$	0.0668	0.6035	0.6245	0.9286	1			
$cv(averages)$	0.09990	0.4867	0.7292	0.0169	0.2973	1		
$average(cvs)$	-0.5017	0.4840	0.6976	0.6335	0.5525	0.2003	1	
RPD	0.4166	0.3800	0.2853	0.5927	0.6385	0.0575	0.0354	1

during the entire search process, and the final population remains far.



a) *average*(*ini_pop*)), *RPD*.

b) *average*(*fin_pop*)), *RPD*.

c) *average*(*averages*)), *RPD*.

d) *cv*(*averages*), *RPD*.

Figure 9.6: Scatter plots of the EGGA convergence.

Finally, to reinforce the conclusions obtained from the exploratory data analysis described above, we generated the graph presented in Figure 9.7 with the indexes that exhibit the stronger relations and allow understanding the EGGA optimization process for the $R||C_{max}$ problem. This diagram includes all the indexes that intervene in the relations studied above and other interest indexes that show significant information about the difficulty of the $R||C_{max}$ instances and the algorithmic behavior of the

EGGA. The diagram comprises: (1) the six more relevant $R||C_{max}$ characteristics: $mach\_corre$, job_corre, q, $cv(min(p_{ij}))$, n, and $multiplicity(lowest)$; (2) the eight measures that provide the most significant information of the EGGA algorithmic behavior, including four measures to analyze the diversity: $cv(cvs)$, $cv(ini\_pop)$, $cv(fin\_pop)$, and $average(cvs)$; as well as four measures to examine the convergence: $cv(averages)$, $average(averages)$, $average(fin\_pop)$, and $average(ini\_pop)$; and (3) the final performance measures RPD and g.

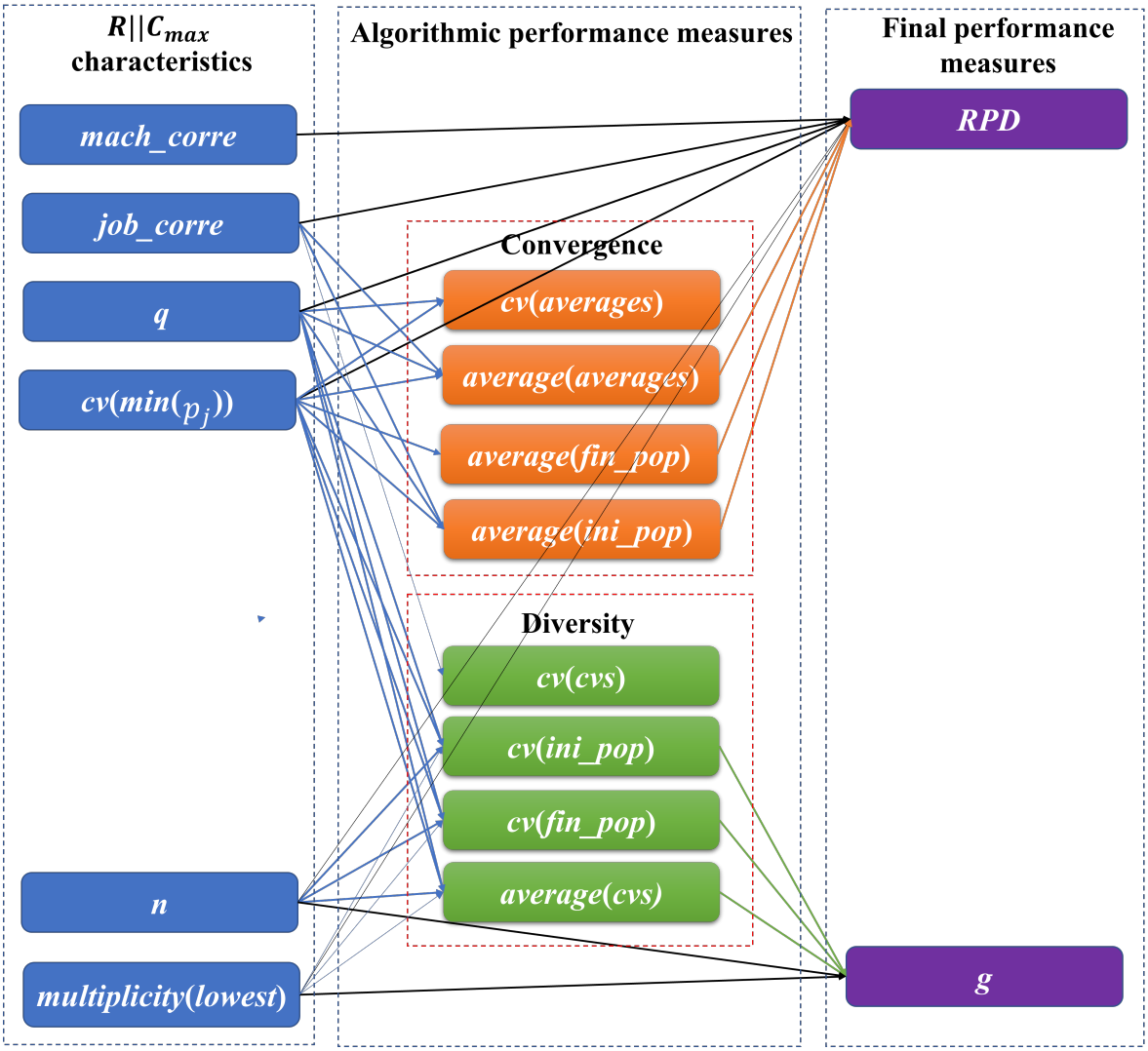


Figure 9.7: Diagram with the relations between the $R||C_{max}$ instances, the EGGA algorithmic behavior, and its final performance.

In the graph presented in Figure 9.7, we emphasize that the six $R||C_{max}$ characteristics are directly related to the EGGA final performance, including the correlation in the distribution of the machines processing times $mach\_corre$ and the jobs processing times job_corre; the dispersion of all the processing times, measured from the coefficient of variation of the minimum processing times of the jobs $cv(min(p_j))$ and the quotient q of the maximum processing time of an instance by the minimum; the size of the instances, analyzed from the number of jobs n; and the structure of the jobs, examined

with the multiplicity of the minimum processing times of the jobs $multiplicity(lowest)$. Furthermore, we highlight in this graph that most of the diversity measures, i.e., $cv(ini\_pop)$, $cv(fin\_pop)$, and $average(cvs)$, are related to the number of generations g used to find the best solution for the complete search. This behavior makes sense since, depending on the EGGA's abilities to handle diversity, it can fully exploit the use of available generations, or it can get stuck in local optima. In contrast, most of the measures to analyze convergence i.e., $average(ini\_pop)$, $average(fin\_pop)$, and $average(averages)$, are more related to the quality of the solutions RPD. This phenomenon is because as EGGA presents a better ability to exploit the search space, it finds better quality solutions. Finally, we emphasize in this graph a relation showing that the $R||C_{max}$ characteristics that affect the way EGGA handles diversity are the correlation in the jobs job_corre, the dispersion of the minimum processing times of the jobs $cv(min(p_{ij}))$, and the quotient q of the maximum processing time of an instance by the minimum. On the other hand, the characteristics that promote the convergence to a greater extent are associated with the size of the instances n, the multiplicity of the minimum processing times of the jobs $multiplicity(lowest)$, the correlation in the jobs job_corre, the dispersion of the minimum processing times of the jobs $cv(p_{ij})$, and the quotient q of the maximum processing time of an instance by the minimum.

As a final conclusion of this study, the diagram, the tables, and the 3D scatter plots suggest that the EGGA needs a strategy to generate the initial population closer to the best-known in order to find better solutions, as well as some strategy to maintain a better balance of exploration and exploitation of the search space. The next section details the proposals to improve the EGGA performance and the experimental results.

9.2.4 Phase 4: Explanations of the algorithmic behavior and proposed improvements

In the previous section, the performance relationships obtained from the EGGA experimental analysis revealed that the strategies included in it, for the generation of the initial population and the search space exploration, presented certain inconsistencies and did not allow it to leave local optima and explore other search space regions. The study also suggested that EGGA requires strategies to maintain better control of the convergence of the population during the search process. In this section, we present the main conclusions obtained from the analysis of strategies that define the algorithm structure and behavior. The EGGA main strategies are examined, identifying the causes of the algorithmic behavior and proposing areas for improvement. The knowledge obtained is used to redesign the algorithm structure and obtain a new version that exhibits a better performance.

Population initialization

The tabular and graphical analysis of the population initialization revealed that EGGA needs to start from an initial population with similar characteristics and as close to the best-known solution as possible to locate better solutions. Derived from this, in this

section, we introduce a set of strategies to initialize the population in search of a strategy capable of generating solutions closer to the best-known solution. The three proposed strategies are called $Fastest\text{-}lb$, $Two\text{-}faster$, and $Two\text{-}faster\text{-}lb$.

As its name implies, the algorithm $Fastest\ lb$ assigns each job j to the machine i that processes it fastest, as long as it does not exceed the lower bound lb. Algorithm 10 shows the $Fastest\ lb$ procedure, which receives as input the set of $jobs[]$ to assign, the set of available $machines[]$, the matrix $data[][]$ with the processing time required by each machine i to process each job j, the vector $fastest[]$ with the machine i that processes each job fastest, and the lower bound lb. lb is equal to the sum of the minimum processing times for each job $Min(p_{ij})$ divided by the number of jobs n. The $Fastest\ lb$ procedure is as follows, it uses four variables: $vector[][]$ to store the set of jobs assigned to each machine, $C_i[]$ to indicate the processing time each machine requires to process its jobs, $free\_jobs[]$ to save jobs that cannot be assigned because they exceed lb, and $permuted\_jobs$ which stores a permutation of jobs $Permutate(jobs)$(lines 1-4). Next, the $permuted\_jobs[]$ are traversed (line 5). Thus, the function $lb\_verification()$ (line 6) is applied to each job j to verifies if the fastest machine i to processes job j can process it without exceeding lb. If machine i can process the job j, $Fastest\_lb()$ returns the machine i, job j is assigned to the machine i and its processing time C_i is updated (lines 7-9). Otherwise, it returns -1, and the job j is added to the array of $free\_jobs$ (lines 9 and 10).

After going through all the $Permutate(jobs)$, $Fastest\ lb$ iterates through the $free\_jobs[]$ (line 14). Thus, the function $Min()$ is applied to each job j, which returns the machine i that processes j faster. In this way, job j is added to machine i and its C_i is updated (line 15-17).

On the other hand, the algorithm $Two\ Fastest$ assigns each job j to one of the two-fastest machines to process it. Algorithm 11 presents the $Two\ fastest$ procedure, which receives as input the sets of $jobs[]$ and available $machines[]$, the processing times $data[][]$, and the vectors with the $fastest[]$ machine and $second-fastest[]$ machine to process each job, respectively. The $Two\ fastest$ strategy uses three variables: $vector[][]$ to store the solution, $C_i[]$ to save the processing time each machine requires to process its assigned jobs, and $permuted\_jobs$ which stores a permutation of the jobs $Permutate(jobs)$(lines 1-3). In this way, $Two\ fastest$ traverses the $permuted\_jobs[]$ (line 4). Thus, it applies the function $Best()$ to each job j, that return the machine i that process faster job j, considering only the two fastest machines (line 5). Next, it appends the job j to the machine i. Finally, it updates the processing time C_i of the machine i (lines 6 and 7).

Similarly, the $Two\ Fastest\ lb$ algorithm assigns each job j to one of the two-fastest machines to processes it, as long as it doesn't exceed the mentioned above lower bound lb. Algorithm 12 describes the $Two\ fastest\ lb$ procedure, which also receives $jobs[]$, $machines[]$, $data[][]$, $fastest[]$, $second-fastest[]$, and lb as input. The $Two\ fastest\ lb$ procedure employees four variables: $vector[][]$ to store the solution, $C_i[]$ to save the machine processing times, $free\_jobs[]$ to collect jobs that exceed lb, and $permuted\_jobs$ which stores a permutation of the jobs $Permutate(jobs)$(lines 1-4). In this way, $Two\ fastest\ lb$ goes through all the $permuted\_jobs[]$ (line 5). Thus, it uses the function $Best-lb()$ to each job j, that returns the machine i that process faster job

Algorithm 10 Fastest lb

Input: $jobs[]$, $machines[]$, $data[][]$, $fastest[]$, and lb.
Output: A solution.

```
vector=[][];
C_i=[];
free_jobs=[];
permuted_jobs[] = Permutate(jobs[]);
for job j in permuted_jobs[] do
    i = lb_verification(C_i, fastest[j], lb);
    if i ≠ −1 then
        Append j to vector[i][];
        Update C_i[i];
    else
        Append j to free_jobs[];
    end if
end for
for job j in free_jobs[] do
    i = Min(C_i, fastest[j]);
    Append j to vector[i][];
    Update C_i[i];
end for
```

Algorithm 11 Two fastest

Input: $jobs[]$, $data[][]$, $fastest[]$, and $second - fastest[]$.
Output: A solution S.

```
vector=[][];
C_i=[];
permuted_jobs[] = Permutate(jobs[]);
for job j in permuted_jobs[] do
    i = Best(C_i, fastest[j], second − fastest[j]);
    Append j to vector[i];
    Update C_i[i];
end for
```

j, as long as it doesn't exceed lb, considering only the two fastest machines. Otherwise, it returns -1 (line 6). If machine i can process job j (line 7), job j is assigned to machine i and its processing time C_i it is updated (lines 8 and 9). Otherwise, the job j is added to the $free\_jobs[]$ vector (line 11). After traversing all the $Permutate(jobs)$, $Two\ fastest\ lb$ iterates through the $free\_jobs[]$ (line 14). Thus, the function $Min()$ is applied to each job j, which returns the fastest machine i to processes job j (line 15). Next, the job j is added to machine i (line 16). Finally, the processing time C_i of machine i is updated (line 17).

Algorithm 12 Two fastest lb

Input: $jobs[]$, $machines[]$, $data[][]$, $fastest[]$, $second-fastest[]$, and lb.
Output: A solution.

1: $vector$=[][];
2: C_i=[];
3: $free\_jobs$=[];
4: $permuted\_jobs[] = Permutate(jobs[])$;
5: **for** job j in $permuted\_jobs[]$ **do**
6: $\quad i = Best-lb(C_i, fastest[j], second-fastest[j])$;
7: $\quad$ **if** $i \neq -1$ **then**
8: $\quad\quad$ Append j to $vector$[i];
9: $\quad\quad$ Update C_i[i];
10: $\quad$ **else**
11: $\quad\quad$ Append j to $free\_jobs[]$;
12: $\quad$ **end if**
13: **end for**
14: **for** job j in $free\_jobs[]$ **do**
15: $\quad i = Min(C_i, fastest[j])$;
16: $\quad$ Append j to $vector$[i];
17: $\quad$ Update C_i[i];
18: **end for**

In order to analyze the performance of the proposed strategies, we conducted an experimental study using the following criteria. We run three EGGA variants for $R||C_{max}$, one with each strategy described above, referred to as EGGA Fastest, EGGA Two-Fastest, and EGGA Two-fastest-lb. The performance of each algorithm is evaluated over the 1400 benchmark instances. To promote a fair comparison, the effectiveness and efficiency of the three EGGA variants are compared by using the same parameter configuration. Population size $|P| = 100$; number of individuals selected for the crossover $n_c = 28$; number of individuals selected for the mutation $n_m = 81$; elite population size $|B| = 12$; Life expectancy $life\_span$=8, and maximal number of generations $max\_gen = 500$. In this way, we analyze the strengths and weaknesses of each population initialization strategy, distinguishing the quality of the solutions found by each EGGA variant and their ability to escape from local optima. Finally, for each instance, a single execution of the algorithms is run with the same initial seed for the random number generator. Thus, we compare the performance of the initialization strategies based on the RPD from each EGGA variant to CPLEX.

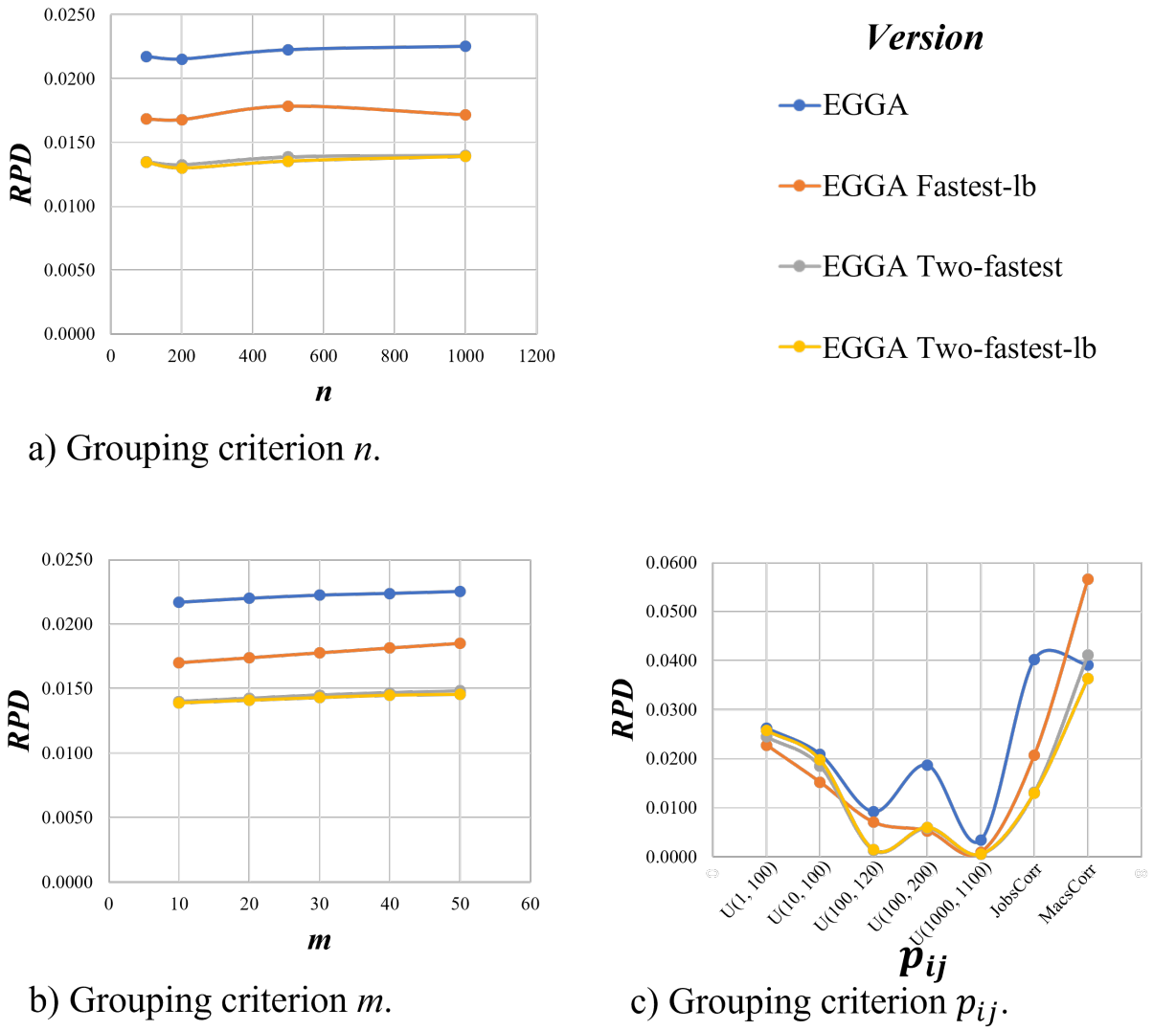
Table 9.8: Comparison of EGGA variants with different population initialization strategies: EGGA Fastest-lb, EGGA Two-fastest, and EGGA Two-fastest-lb using RPD.

Instance set		EGGA	EGGA Fastest-lb	EGGA Two-fastest	EGGA Two-fastest-lb
n	100	0.0217	0.0168	**0.0135**	**0.0135**
	200	0.0215	0.0168	0.0132	**0.0130**
	500	0.0223	0.0178	0.0139	**0.0135**
	1000	0.0225	0.0172	0.0140	**0.0139**
m	10	0.0217	0.0170	0.0140	**0.0139**
	20	0.0220	0.0174	0.0143	**0.0141**
	30	0.0222	0.0178	0.0145	**0.0143**
	40	0.0224	0.0182	0.0147	**0.0145**
	50	0.0225	0.0185	0.0148	**0.0145**
p_{ij}	$U(1,100)$	0.0262	**0.0228**	0.0245	0.0257
	$U(10,100)$	0.0209	**0.0152**	0.0185	0.0198
	$U(100,120)$	0.0092	0.0070	**0.0013**	0.0014
	$U(100,200)$	0.0187	**0.0053**	0.0058	0.0060
	$U(1000,1100)$	0.0034	0.0009	**0.0005**	**0.0005**
	$JobsCorr$	0.0402	0.0207	0.0131	**0.0130**
	$MacsCorr$	0.0392	0.0566	0.0411	**0.0363**
1400 *instances*		0.0225	0.0184	0.0150	**0.0147**

Table 9.8 contains the experimental results values obtained by each EGGA. For a more detailed study, we compare the performance of the algorithms using four different criteria to group the 1400 instances: the number of jobs n, the number of machines m, the distribution of the processing times p_{ij}, and the 1400 instances together. The first and second columns indicate the criteria used to group the test instances: n, m, p_{ij}, and the 1400 instances. The remaining columns contain the average RPD obtained by each EGGA for the four grouping criteria, highlighting the best results in bold. Each EGGA is identified with the population initialization strategy that they use as EGGA Fastest-lb, EGGA Two-fastest, and EGGA Two-fastest-lb.

The experimental results in Table 9.8 suggest that the most suitable population initialization strategy is $Two\text{-}fastest\text{-}lb$ with an average RPD of 0.0147 since it improved the EGGA performance about 35%. Furthermore, Table 9.8 suggests that if we consider only the distribution of processing times p_{ij}, the initialization strategy $Fastest\text{-}lb$ is the best option to address instances where $q = max(p_{ij})/min(p_{ij})$ is greater than or equal to 2. That is, the instances in the sets $U(1,100)$, $U(10,100)$, and $U(100,200)$, while in the rest of the instance sets, it is better to use the $Two\text{-}fastest\text{-}lb$ strategy.

Additionally, Figure 9.8 graphically displays the performance of the four EGGA variants (the EGGA under investigation and the three EGGAs with the proposed population initialization strategies). Each graph shows the average performance of the four



a) Grouping criterion n.

b) Grouping criterion m.

c) Grouping criterion p_{ij}.

Figure 9.8: Graphical comparison of the proposed population initialization strategies $Fastest\text{-}lb$, $Two\text{-}fastest$, and $Two\text{-}fastest\text{-}lb$ using the grouping criteria n, m, and p_{ij}.

algorithms in the 1400 instances, grouped with respect to one criterion: n, m, and p_{ij}. These graphs allow validating the conclusions obtained from the tabular analyses, where the strategy $Two\text{-}fastest\text{-}lb$ presented the best results.

Rearrangement heuristics for mutation

In addition to the need for an improvement to the population initialization strategy, the study of the EGGA optimization process also revealed that it needs a change on its structure and behavior to maintain a better balance of exploration and exploitation of the search space. Derived from this, in this section, we introduce a set of strategies to enhance the mutation operator. We decided to work with this operator because the knowledge obtained from the optimization study of each EGGA component allowed

identifying that it has one of the most important roles in the optimization process. Moreover, we observed that the good or bad behavior of the mutation impacts directly on the EGGA performance. Finally, we noticed that the designed mutation operator still has several areas of opportunity. Therefore, in this section, we introduce a set of strategies to improve the mutation algorithmic behavior. In this way, we generated four EGGA variants, one for each proposed rearrangement heuristic, referred to as EGGA Interchange, EGGA 2-Best, EGGA 4-Best, and EGGA Injection.

The proposals are focused on improving the rearrangement strategy Assemble and are applied when it cannot perform an insertion or interchange without affecting the quality of elite solutions, i.e., the best $|B|$ solutions of the current population. In this manner, EGGA Interchange improves the rearrangement strategy Assemble as follows. If Assemble cannot modify a solution, it performs the exchange that least affects the solution. Similarly, EGGA 2-Best and EGGA 4-Best incorporate strategies that release two and four jobs from each machine, respectively, and reinsert them with the heuristic Best() when Assemble cannot modify a solution. Finally, EGGA Injection replaces the solutions that the rearrangement heuristic cannot modify with one solution generated with the Two-fast-lb strategy.

To analyze the performance of the proposed strategies, we conducted an experimental study using the before-mentioned criteria. That is, we run the four EGGA variants and evaluate them over the 1400 benchmark instances. To promote a fair comparison, the effectiveness and efficiency of the four EGGA variants are compared by using the same parameter configuration. $|P| = 100$; $n_c = 28$; $n_m = 81$; $|B| = 12$; $life\_span$=8, and $max\_gen = 500$. Finally, for each instance, a single execution of the algorithms is run with the same initial seed for the random number generator. In this way, we compare the performance of the proposed strategies based on the RPD from each EGGA to CPLEX.

Table 9.9 contains the experimental results values obtained EGGA, EGGA Interchange, EGGA 2-Best, EGGA 4-Best, and EGGA Injection. Like in Table 9.8, for a comprehensive study, the instances are grouped with respect to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij} of the instances, and the complete benchmark (1400 instances). The first two columns denote the criteria used to group the test instances: n, m, p_{ij}, and the 1400 instances. On the other hand, the remaining columns contain the average RPD obtained by each EGGA variant for the four grouping criteria, respectively, highlighting the best results in bold.

The experimental results in Table 9.9 indicate that the EGGA Injection has a high performance in the most difficult instances $U(1, 100)$, $U(10, 100)$, and $MacsCorr$, improving the results of the state-of-the-art for the sets $U(1, 100)$, $U(10, 100)$. This behavior is interesting because it indicates that, in problems where the quotient of variation of the maximum processing time by the minimum q is high, it is necessary to add new solutions during the optimization process to generate new search directions. On the other hand, Table 9.9 also indicates that the EGGA Interchange showed the best average performance since it obtained better solutions in the instances where EGGA injection did not, i.e., $U(100, 120)$, $U(100, 200)$, $U(1000, 1100)$, and $JobsCorr$. From this study we conclude that if the instances have a high value of q or have correlated machines, it is better to use a strategy that incorporates a lot of diversity, such as

Table 9.9: Comparison of EGGA variants with different strategies to handle diversity using RPD: EGGA Interchange, EGGA 2-Best, EGGA 4-Best, and EGGA Injection using RPD.

Instance set		**EGGA**	**EGGA Interchange**	**EGGA 2-Best**	**EGGA 4-Best**	**EGGA Injection**
n	100	0.0136	0.0124	0.0128	0.0125	**0.0122**
	200	0.0130	0.0124	0.0126	**0.0123**	0.0125
	500	0.0135	0.0130	0.0131	**0.0128**	0.0142
	1000	0.0133	**0.0130**	0.0134	0.0131	0.0198
m	10	0.0137	**0.0130**	0.0135	0.0132	0.0149
	20	0.0139	**0.0134**	0.0138	0.0134	0.0148
	30	0.0142	**0.0137**	0.0140	0.0137	0.0142
	40	0.0143	**0.0138**	0.0142	0.0139	0.0141
	50	0.0145	0.0140	0.0143	0.0140	**0.0136**
p_{ij}	$U(1,100)$	0.0239	0.0237	0.0263	0.0249	**0.0150**
	$U(10,100)$	0.0205	0.0167	0.0168	0.0159	**0.0149**
	$U(100,120)$	0.0025	0.0014	0.0014	0.0014	0.0146
	$U(100,200)$	0.0057	0.0050	0.0050	**0.0046**	0.0146
	$U(1000,1100)$	0.0008	0.0004	0.0004	**0.0004**	0.0146
	$JobsCorr$	0.0128	**0.0110**	0.0115	0.0113	0.0151
	$MacsCorr$	0.0347	0.0410	0.0405	0.0407	**0.0151**
1400 instances		0.0144	**0.0142**	0.0145	0.0142	0.0147

Injection, but if the values of q of the instances are small, it is better to use a strategy that incorporates diversity in a more measured way, such as interchange. Finally, as the average performance of all variants was very similar, we will select the EGGA Interchange, since it does not represent a significant extra computational cost. Thus, as this is the last improvement to the EGGA studied, the EGGA Interchange will be referred to as the Final GGA (FGGA).

9.3 Conclusions of the characterization

In this chapter, we presented a characterization study of a set of $R||C_{max}$ instances and the EGGA algorithmic behavior. The experimental results suggested that the instances of $R||C_{max}$ are more difficult when they have a larger number of jobs, a high quotient of the largest and the shortest processing times, correlated machines, and much variability of the minimum processing times of the jobs. Furthermore, these results allowed identifying some paths of work to improve the EGGA performance. In this fashion, we observed that the EGGA could be improved by replacing its strategy to generate the initial population with a heuristic able to generate solutions closer to the best-known solutions, as well as incorporating a strategy to provide EGGA the capability to explore and exploit the search space efficiently. The knowledge obtained from this work was used to design a new population initialization strategy for the EGGA and to improve the mutation operator by incorporating a rearrangement heuristic to modify the solutions that the mutation operator cannot alter by itself. The improvement study for the EGGA revealed that in problems with a high quotient q or correlated machines, it is necessary to use strategies that promote a lot of diversity, as the injection strategy. In contrast, in problems with a low quotient q or correlated jobs, it is better to incorporate a strategy promoting diversity in a more controlled way, as the interchange strategy. Finally, thanks to knowledge obtained from the characterization study and the proposed enhancements to EGGA, an improvement rate of about 35% was obtained. These results demonstrate the importance of the characterization and analysis of the optimization process to solve a problem. Chapter 10 presents a study to analyze the strengths and weaknesses of the final version of the EGGA, referred to as FGGA that includes the intelligent strategies introduced in this chapter.

Chapter 10

Performance analysis of the FGGA for $R||C_{max}$

This chapter presents the main computational experiments used to analyze different aspects of the FGGA performance for the NP-hard combinatorial optimization grouping problem $R||C_{max}$. This study covers from the initial GGA version to its final improvement (FGGA), presented in the previous section, to analyze the way in which its performance evolved until reaching the current version. Moreover, it includes a robustness study to examine the performance of FGGA for $R||C_{max}$ by conducting different runs with different seeds. Finally, it presents a set of experiments to explore the FGGA capabilities in a long term by performing different runs with distinct numbers of generations. The experimental results show the good performance of the FGGA, demonstrating the usefulness of the method used in this work to improve the performance of metaheuristic algorithms.

10.1 Components of FGGA

This section includes a general description of the FGGA components, standing out its main characteristics. After performing the extensive experimentation detailed in the previous sections, the components of the FGGA are as follows.

- Initialization strategy: Two-fastest-lb that uses a permutation of the jobs to assign them to one of the two machines that process them faster until reaching a lower bound lb, the rest of the jobs are assigned with the Min() allocation strategy (see Chapter 9).
- Crossover operator: IE-Two machines that organize the parent machines based on their processing times C_i, rearranging the tied machines according to the number of jobs assigned to them N_{jobs}. During the transmission, both parents transmit their machines, giving priority to the one with the lowest C_i and, if necessary, breaking the tie randomly. Finally, the crossover operator removes repeated jobs to avoid repeated genetic material and generates only one child (see Chapter 6).

- Mutation operator: 2-Items Reinsertion that randomly chooses two jobs from two different machines to release them and later reinsert them with the Min() allocation heuristic (see Chapter 7).
- Rearrangement strategy: Assemble that works based on insertion and interchange operations. The rearrangement process is only applied if, after releasing and reinserting the jobs, the genetic material of the mutated solution has not been modified (see Chapter 7).
- Reproduction strategy: Ranking BRW (best, repeated, and worst) that use a strategy to arrange the solutions from best to worst, to later rearrange the population, placing at the end the solutions that meet the following three criteria: (1) the same makespan C_{max}, (2) the same number of machines with $C_i = C_{max}$, and (3) the same average processing time Average(C_i). In this way, the reproduction technique selects the first individuals of the ordered population (the best ones without repeated solutions) for crossover and introduces the offspring to the population, replacing the repeated (or similar) solutions and later the solutions with the worst fitness. Similarly, this reproduction technique mutates the first solutions of the ordered population (the best). To avoid the loss of good solutions, Ranking BRW clones elite solutions before mutating them, replacing repeated (or similar) solutions and then the solutions with the worst fitness by the clones (see Chapter 8).
- Parameter setting: we use the covering array approach proposed by Quiroz-Castellano *et al.* [11] to configure FGGA. As a result, the final configuration is as follows. Population size $|P| = 100$; number of individuals selected for the crossover $n_c = 28$; number of individuals selected for the mutation $n_m = 81$; elite population size $|B| = 12$; Life expectancy $life\_span$=8, and maximal number of generations $max\_gen = 500$.

The following sections present a set of experimental studies to analyze different aspects of the FGGA, described above. In this way, we look for demonstrating the usefulness of the characterization approach used in this work.

10.2 Evolution of the GGA performance

Although many GGA variants were developed during this research project in search of opportunity niches, the following three versions set the course for this research, and they allow seeing the way in which the GGA performance improved until reaching the current version.

- The initial GGA, which is an adaptation of the state-of-the-art Grouping Genetic Algorithm with Controlled Genes Transmission (GGA-CGT) introduced by Quiroz-Castellanos *et al.* to solve the Bin Packing problem [11] (see Chapter 4).
- The Enhanced GGA (EGGA), generated from the study and individual improvement of each component of the Initial GGA, including the strategy for

initial population (see Chapter 5), crossover (see Chapter 6), mutation (see Chapter 7), and the reproduction technique (see Chapter 8).

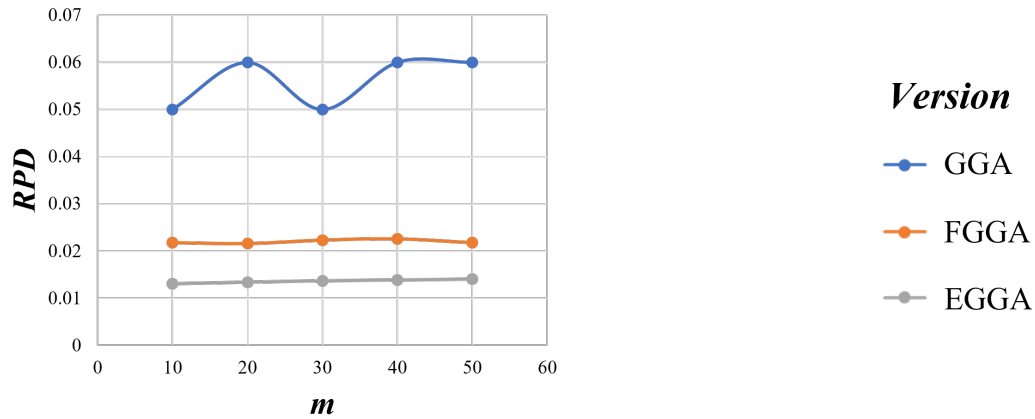
– The Final GGA (FGGA) resulted from the observations and improvements made by using the algorithmic behavior characterization approach (see Chapter 9).

Table 10.1 shows the experimental results of the study conducted to compare GGA, EGGA, and FGGA. For a comprehensive analysis, we distributed the RPD values reached by the three GGA versions in groups of instances sorted according to the number of machines m, the number of jobs n, the processing time of the jobs p_{ij}, and the 1400 instances together. The first and second columns indicate the criteria used to group the test instances, and the remaining columns contain the average RPD obtained by each GGA version, highlighting in bold the metaheuristic algorithm with the lowest average RPD for each set.

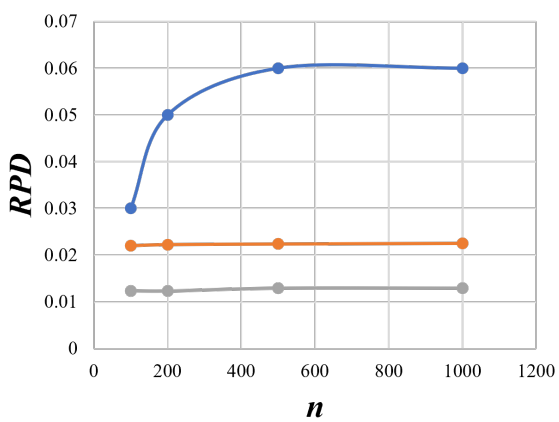
Table 10.1: Comparison of GGA, EGGA, and FGGA using RPD.

Instance set		**GGA**	**EGGA**	**FGGA**
n	100	0.0659	0.0211	**0.0124**
	200	0.0655	0.0209	**0.0124**
	500	0.0657	0.0212	**0.0130**
	1000	0.0688	0.0220	**0.0130**
m	10	0.0683	0.0211	**0.0130**
	20	0.0683	0.0213	**0.0134**
	30	0.0683	0.0215	**0.0137**
	40	0.0683	0.0217	**0.0138**
	50	0.0683	0.0218	**0.0140**
p_{ij}	$U(1, 100)$	0.1027	0.0278	**0.0237**
	$U(10, 100)$	0.1119	0.0206	**0.0167**
	$U(100, 120)$	0.0256	0.0094	**0.0014**
	$U(100, 200)$	0.0829	0.0175	**0.0050**
	$U(1000, 1100)$	0.0121	0.0031	**0.0004**
	$jobsCorr$	0.0586	0.0374	**0.0110**
	$MacsCorr$	0.0955	**0.0365**	0.0410
1400 instances		0.0699	0.0218	**0.0142**

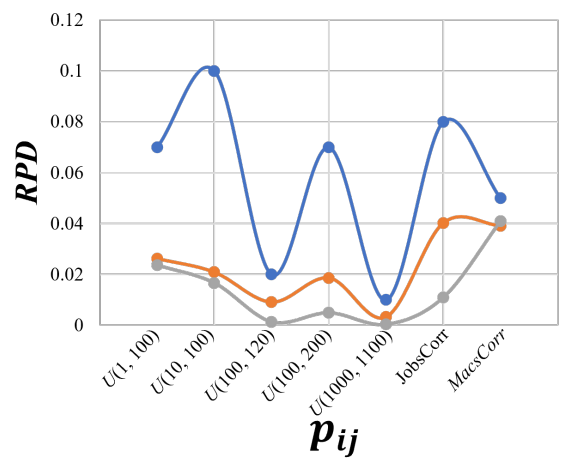
Table 10.1 indicates that the rate of improvement between the initial version and the improved version is 62%. While the rate of improvement between the initial version and the final version is 76%. Furthermore, Figure 10.1 graphically displays the performance evolution of the GGA. Each graph shows the average performance of the three GGA variants in the 1400 instances, grouped with respect to three criteria. In this way, Figures 10.1a, 10.1b, and 10.1c show the way the GGA behavior change according to the number of jobs n, the number of machines m, and the distribution of the processing times p_{ij}, respectively. In each graph, the $x-$axis represents the criteria used to group the instances, and the $y-$axis indicates the RPD achieved for each set of instances.



a) Number of machines m.



b) Number of jobs n.



c) Distribution of the processing time of the jobs p_{ij}.

Figure 10.1: Graphical comparison of the GGA, EGGA, and FGGA using the grouping criteria n, m, and p_{ij}.

Therefore, the blue, orange, and gray lines represent the average RPD reached for each set by the GGA, EGGA, and FGGA, respectively. These graphs allow validating the conclusions obtained from the tabular analyses and showing how the performance of the GGA improved with each enhancement performed. From Figure 10.1 emerges that the initial GGA performance is affected by the number of machines (Figure 10.1a) and the number of jobs (Figure 10.1b), decreasing its performance when increasing the number of jobs. In contrast, the EGGA and the FGGA are more robust regarding the number of machines and jobs, because both incorporate heuristic strategies designed by using knowledge of the problem domain. This knowledge was generated from the study of the algorithmic behavior of each GGA component (see Chapters 5, 6, 7, and 8) and the characterization of the C_{max} problem and the EGGA optimization process (see

Table 10.2: p-Values of the Wilcoxon test for the initial GGA and the FGGA.

Instance Set		p-Value
n	100	8.17E-44
	200	4.94E-69
	500	1.27E-84
	1000	5.08E-82
m	10	2.86E-76
	20	1.18E-63
	30	2.24E-45
	40	8.85E-40
	50	1.15E-42
p_{ij}	$U(1, 100)$	6.06E-45
	$U(10, 100)$	4.93E-66
	$U(100, 120)$	3.32E-56
	$U(100, 200)$	3.63E-65
	$U(1000, 1100)$	4.42E-66
	$JobsCorr$	3.75E-65
	$MacsCorr$	1.52E-29
1400 Instances		1.17E-253

Chapter 10). Finally, Figure 10.1c reveals that the distribution of processing times p_{ij} has an impact on the performance of the three GGA versions, being the most difficult classes $U(1, 100)$, $U(10, 100)$ and $MacsCorr$.

Additionally, we applied the Wilcoxon rank-sum test to assess whether the differences in the RPD achieved by the initial GGA and the FGGA for the 1,400 test instances are statistically significant. The Wilcoxon rank-sum is a non-parametric test that compares two algorithms without assuming a Normal distribution of the results samples, even for small sample sizes [401]. Table 10.2 presents the results obtained by the Wilcoxon rank-sum for the RPD values reached by both algorithms in the benchmark considered with a 95%-confidence level. For a comprehensive comparison, we generated a hypothesis test for the RPD achieved by both GGAs in groups of instances sorted according to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij} of the instances, and the complete benchmark (1400 instances). In this way, the first column indicates the criterion used to compare the algorithms, the second one contains the classes covered for each grouping criterion, and the last column indicates the p-Values obtained by the Wilcoxon test.

Table 10.2 indicates that the FGGA is indeed statistically better than the initial GGA considering the RPD that they reached for the test benchmark for all the groups of instances considered since all p-Values are less than the level of significance $\alpha = 0.05$.

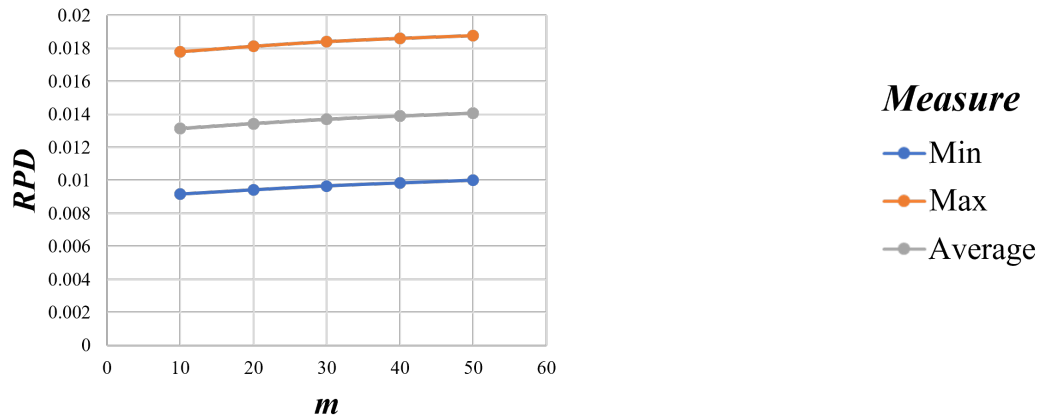
Table 10.3: Comparison of the FGGA performance with ten different seeds using RPD.

Instance set		Min	Max	Average	Std
n	100	0.0087	0.0170	0.0125	0.0053
	200	0.0086	0.0166	0.0123	0.0044
	500	0.0095	0.0169	0.0129	0.0032
	1000	0.0093	0.0175	0.0131	0.0053
m	10	0.0092	0.0178	0.0131	0.0052
	20	0.0094	0.0181	0.0134	0.0052
	30	0.0097	0.0184	0.0137	0.0052
	40	0.0098	0.0186	0.0139	0.0052
	50	0.0100	0.0188	0.0141	0.0052
p_{ij}	$U(1, 100)$	0.0153	0.0353	0.0248	0.0101
	$U(10, 100)$	0.0113	0.0236	0.0168	0.0044
	$U(100, 120)$	0.0008	0.0021	0.0014	0.0007
	$U(100, 200)$	0.0030	0.0071	0.0049	0.0008
	$U(1000, 1100)$	0.0002	0.0006	0.0004	0.0001
	$JobsCorr$	0.0064	0.0158	0.0107	0.0027
	$MacsCorr$	0.0341	0.0481	0.0406	0.0023
1400 instances		0.0101	0.0190	0.0142	0.0051

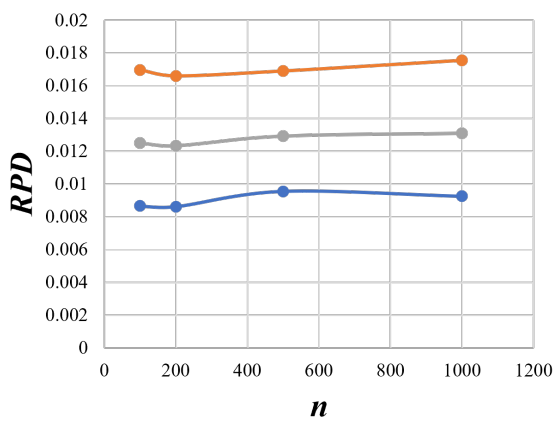
10.3 FGGA robustness test

After analyzing the FGGA performance with the improvements made, we conduct an experimental study with 10 runs using different seeds to know its robustness. Table 10.3 includes the experimental results. For a fair comparison, the test instances are grouped with respect to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij} of the instances, and the complete benchmark (1400 instances). The criteria used to group the test instances are indicated in columns one and two: n, m, p_{ij}, and the 1400 instances. The remaining columns indicate the best solution found (Min), the worst (Min), the mean (Average), and the standard deviation (Std) based on the RPD. Additionally, Figure 10.2 graphically displays the best (Min), worst (Max), and mean (Average) RPD of the solutions found by the FGGA in the ten runs. Each graph shows these three measures (Min, Max, and Average) from the ten runs of the FGGA with different seeds in the 1400 instances, grouped with respect to three criteria. In this way, Figures 10.2a, 10.2b, and 10.2c show the best (Min), worst (Max), and mean (Average) GGA algorithmic behavior according to the number of jobs n, the number of machines m, and the distribution of the processing times p_{ij}, respectively. In each graph, the $x-$axis represents the criteria used to group the instances, and the $y-$axis indicates the RPD with each measure (Min, Max, and Average) for each set of instances. Therefore, the lines blue, orange, and gray represent the average RPD for the best (Min), worst (Max), and mean (Average) solution, respectively.

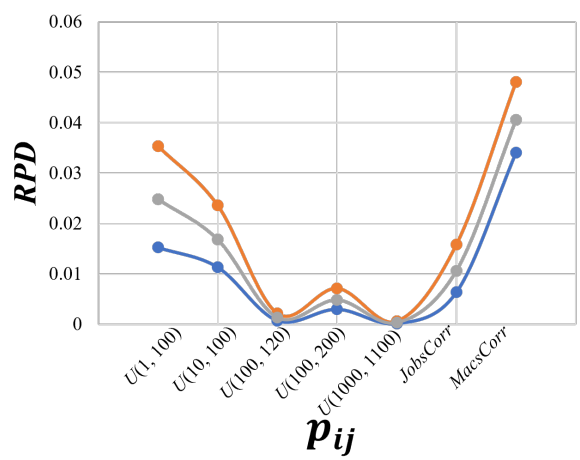
The tabular and the graphical analyses conducted from Table 10.3 and Figure 10.2



a) Number of machines m.



b) Number of jobs n.



c) Distribution of the processing time of the jobs p_{ij}.

Figure 10.2: Graphical comparison of the FGGA performance with ten different seeds based on RPD.

allow observing that the FGGA algorithmic behavior is quite stable in the instances regardless of the number of machines (Figure 10.2a) or jobs (Figure 10.2b). However, if we analyze the ten executions with respect to the distribution of the processing times of the instances (Figure 10.2c), we can observe that in some sets like $U(100, 120)$ and $U(1000, 1100)$ the performance of the FGGA is stable. These two sets have in common that the instances have the smaller quotient q of their longest and shortest processing times. Therefore, the instances of the sets $U(100, 120)$ and $U(1000, 1100)$ have values of $q = 1.2$ and $q = 1.1$, respectively. Likewise, these results show that the variations in the FGGA behavior increase as the instances have higher q values. Therefore, FGGA presents the largest variations on its performance in the sets $U(1, 100)$ and $U(10, 100)$, whose quotients are 100 and 10, respectively.

Table 10.4: Comparison of the FGGA performance with the *max_gen* values 500, 1000, 2000, and 10000 using *RPD*.

Instance set		500	1000	2000	10000
n	100	0.0123	0.0089	0.0064	**0.0037**
	200	0.0123	0.0085	0.0061	**0.0034**
	500	0.0129	0.0085	0.0058	**0.0028**
	1000	0.0129	0.0093	0.0068	**0.0040**
m	10	0.0129	0.0093	0.0067	**0.0040**
	20	0.0133	0.0094	0.0068	**0.0041**
	30	0.0135	0.0096	0.0069	**0.0041**
	40	0.0137	0.0096	0.0069	**0.0040**
	50	0.0139	0.0097	0.0069	**0.0040**
p_{ij}	$U(1, 100)$	0.0237	0.0208	0.0174	**0.0147**
	$U(10, 100)$	0.0167	0.0120	0.0105	**0.0075**
	$U(100, 120)$	0.0013	0.0008	0.0005	**0.0002**
	$U(100, 200)$	0.0050	0.0034	0.0026	**0.0014**
	$U(1000, 1100)$	0.0004	0.0002	0.0002	**0.0000**
	JobsCorr	0.0105	0.0068	0.0036	**-0.0009**
	MacsCorr	0.0409	0.0250	0.0143	**0.0058**
1400 instances		0.0141	0.0098	0.0070	**0.0041**

10.4 FGGA long-term execution

Once the robustness of the FGGA was analyzed, in this chapter we investigated how the values of *max_gen* impact on the FGGA performance to know its algorithmic behavior in the long term. Recalling from Chapter 4, the *max_gen* parameter controls the FGGA execution time. Therefore, in this study, we consider the parameter values 500, 1000, 2000, and 10000. Table 10.4 shows the experimental results for the test instances grouped with respect to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij} of the instances, and the complete benchmark (1400 instances). The criteria used to group the test instances are indicated in columns one and two: n, m, p_{ij}, and the 1400 instances. The remaining columns contain the average *RPD* obtained by FGGA with different values of *max_gen*, highlighting the best results in bold.

The results in Table 10.4 prove that the FGGA can reach high-quality solutions of a huge variety of $R||Cmax$ instances. Likewise, it demonstrates that the FGGA performance is still improving as a larger *max_gen* value is considered. Furthermore, from Table 10.4 it is possible to observe that the instances generated with a distribution of the processing times in the ranges $U(1, 100)$ and $U(10, 100)$ and with machines correlated *MacsCorr* still show a high degree of difficulty for the FGGA; while, the instances generated with a distribution of the processing times in the ranges $U(100, 120)$ and $U(1000, 1100)$, and with jobs correlated *JobsCorr* represent a lower degree

of difficulty.

Additionally, Figure 10.3 graphically displays the way the FGGA performance improves as a larger max_gen value is considered. Each graph shows the average performance of the FGGA with a different max_gen value in the 1400 instances, grouped with respect to three criteria. In this way, Figures 10.3a, 10.3b, and 10.3c show the FGGA algorithmic behavior with the different max_gen values according to the number of jobs n, the number of machines m, and the distribution of the processing times p_{ij}, respectively. In each graph, the $x-$axis represents the criteria used to group the instances, and the $y-$axis indicates the RPD achieved with each max_gen value for each set of instances. Therefore, the blue, orange, gray, and yellow lines represent the average RPD reached for each set by the FGGA with 500, 1000, 2000, and 10000 generations, respectively. These graphs allow validating the conclusions obtained from the tabular analyses and graphically showing how the FGGA performance can still improve. This behavior is remarkable compared to the number of iterations required by state-of-the-art population strategies for other grouping problems like Bin Packing [12, 402, 275].

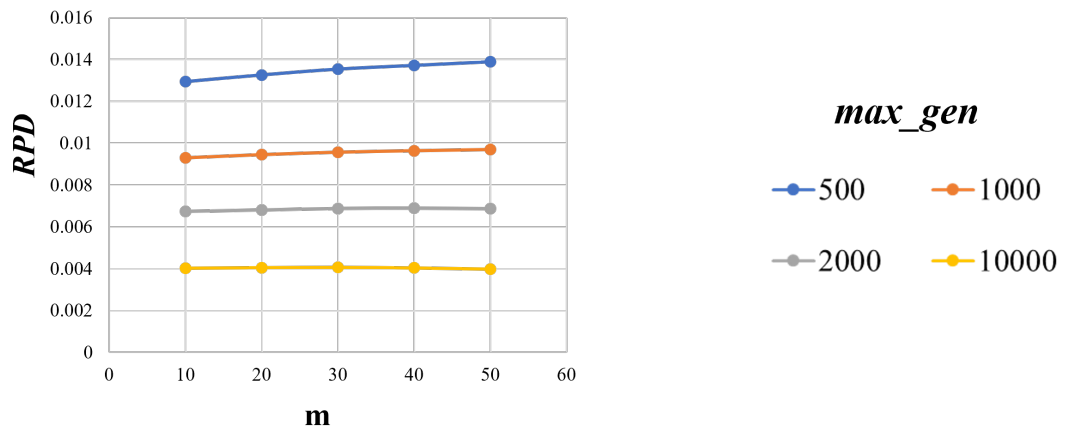
Finally, we perform a second tabular study to identify the instances that FGGA can solve better than CPLEX. Table 10.5 shows the experimental results. For a fair comparison, the test instances are grouped with respect to the number of jobs n, the number of machines m, the distribution of the processing times p_{ij} of the instances, and the complete benchmark (1400 instances). The criteria used to group the test instances are indicated in columns one and two: n, m, p_{ij}, and the 1400 instances. The remaining columns indicate the number of instances that FGGA finds a better solution than CPLEX, highlighting in bold the best results.

Table 10.5 allows us to observe that, by configuring the FGGA with 10000 generations, it can find better solutions than CPLEX in most sets of instances. Furthermore, these results allow reiterating that the sets $U(1, 100)$, $U(10, 100)$, and $MacsCorr$ represent a greater challenge for the FGGA, since it cannot find any solution better than CPLEX in the set $U(1, 100)$ and only finds one solution in the sets $U(10, 100)$ and $MacsCorr$.

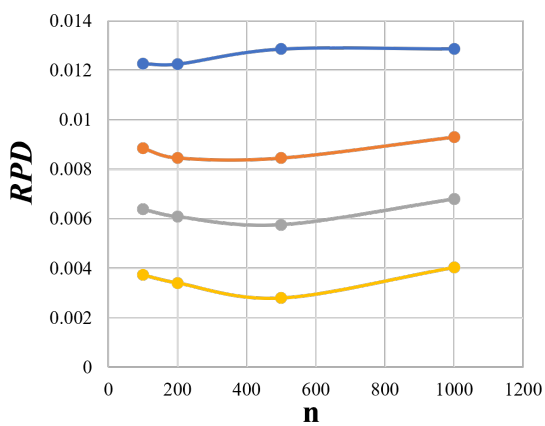
10.5 Comparing FGGA with state-of-the-art procedures

In this section, we present the latest study of the FGGA performance, which consists of comparing its performance against the best state-of-the-art algorithms. We consider the best two-phase algorithm, Partial enumeration of Mokotoff and Jimeno [358]; the best exact method, Recovering Beam Search (RBS) of Ghirardi and Potts [374]; the best local search Iterative Greedy local search of Fanjul-Peyro and Ruiz, referred to as NVST-IG+ [346]; and hybrid method, the Hybrid Tabu Search of Sels et al., referred to as HTS [96]. In this way, we compare their results presented in Section 3.2 against the FGGA results with 10000 generations.

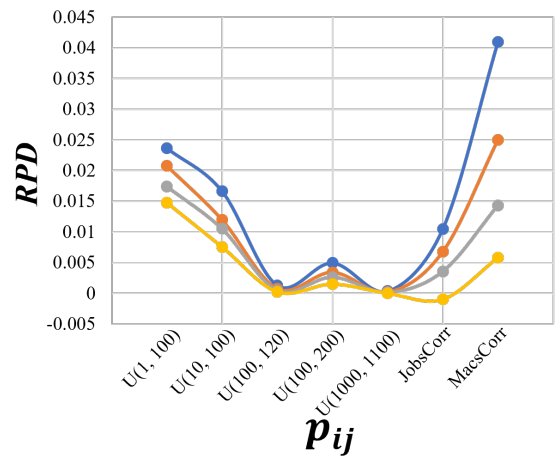
Table 10.6 contains the experimental results. For a comprehensive analysis, we distribute the RPD values reached by the solution methods in groups of instances,



a) Number of machines m.



b) Number of jobs n.



c) Distribution of the processing time of the jobs p_{ij}.

Figure 10.3: Graphical comparison of the FGGA performance with different values of the parameter max_gen using the grouping criteria n, m, and p_{ij}.

sorted according to the distribution of the processing times p_{ij} of the instances, and we also analyze the average RPD for the complete benchmark (1400 instances). The first column indicates the criteria used to group the test instances, and the remaining columns contain the average RPD obtained by each algorithm for each grouping criteria used. The best values are indicated in bold.

Table 10.5: Comparison of the FGGA performance with the *max_gen* values 500, 1000, 2000, and 10000 based on the number of instances that it finds a better solution than CPLEX.

	Instance set	**500**	**1000**	**2000**	**10000**
n	100	2	4	8	**27**
	200	12	17	18	**35**
	500	1	6	18	**36**
	1000	1	4	9	**34**
m	10	0	0	2	**4**
	20	0	2	1	**10**
	30	11	16	19	**31**
	40	3	4	10	**39**
	50	2	9	21	**48**
p_{ij}	$U(1, 100)$	0	0	0	**0**
	$U(10, 100)$	0	0	0	**1**
	$U(100, 120)$	1	1	7	**21**
	$U(100, 200)$	5	6	7	**13**
	$U(1000, 1100)$	7	14	17	**35**
	$JobsCorr$	3	10	21	**61**
	$MacsCorre$	0	0	**1**	**1**
	Total	16	31	53	**132**

Table 10.6: Analysis of the average RPD reached by the state-of-the-art algorithms: Partial, RBS, NVST-IG+, HTS, and FGGA for the 1400 instances.

Instance Set	**Partial**	**RBS**	**NVST-IG+**	**HTS**	**FGGA**
$U(1, 100)$	0.0288	0.0203	**0.0134**	0.0183	0.0147
$U(10, 100)$	0.0131	0.0187	**0.0075**	0.0151	**0.0075**
$U(100, 120)$	0.0033	0.0013	0.0004	**0.0000**	0.0002
$U(100, 200)$	0.0105	0.0081	0.0032	**0.0008**	0.0014
$U(1000, 1100)$	0.0023	0.0018	0.0002	**-0.0001**	0.0000
$JobsCorr$	0.0234	0.0035	0.0048	**-0.0053**	-0.0009
$MacsCorr$	0.0094	0.0236	0.0055	**0.0038**	0.0058
1400 instances	0.0130	0.0110	0.0050	0.0047	**0.0041**

From Table 10.6 can be observed that FGGA reaches better results on solving the 1400 instances because it finds better solutions in the sets $U(1, 100)$ and $U(10, 100)$ than HTS. Likewise, it has a better performance than NVST-IG in the sets $U(100, 120)$, $U(100, 200)$, $U(1000, 1100)$, and $JobsCorr$. This study shows that the sets $U(1, 100)$ and $U(10, 100)$ not only represent a great difficulty for the FGGA, but also for all state-of-the-art algorithms. Among the two main characteristics that share the instances in these sets are (1) that all their processing times p_{ij} are less than or equal

to 100 and (2) that their quotient q of the largest by the shortest processing time are the highest of the benchmark. The set $U(1, 100)$ has a value of q =100 and $U(10, 100)$ a value of q =10. This implies that, for example, for the set $U(1, 100)$ there may be instances with jobs that require up to 100 times more processing time than other ones. In the same way, the time needed by a machine to process a job can be 100 more than the time needed for the other ones.

Chapter 11

Conclusions and future work

This chapter presents the conclusions and final results obtained from the conducted research project, covering the design of the first GGA for $R||C_{max}$, going through the improvements proposed based on the knowledge generated from the set of the systematical study performed to each GGA component in isolation, and ending with the enhancements achieved through the characterization approach implemented. Likewise, it presents suggestions for the development of future work.

11.1 Conclusions

The research work presented in this thesis addressed the problem of building robust and highly effective heuristic strategies that incorporate knowledge of the problem domain to solve the NP-hard grouping problem Unrelated Parallel-Machine Scheduling with Makespan Minimization $R||C_{max}$. To achieve this goal, we presented the first GGA for $R||C_{max}$, based on the GGA-CGT proposed for the Bin Packing problem [11]. As GGA-CGT includes knowledge of the Bin Packing problem domain, the GGA results were good, but they are far from the state-of-the-art algorithms. Therefore, we analyzed the optimization process of each GGA component in isolation, that is, the initialization strategy, the crossover and mutation operators, as well as the reproduction technique. Thus, we identified the aspects that intervene during each GGA sub-process (initialization, crossover, mutation, selection, and replacement) to study and understand how they impact the GGA performance. The knowledge obtained was used to generate intelligent strategies of specific-purpose heuristics for $R||C_{max}$ that were incorporated into the GGA, giving rise to the Enhanced GGA (EGGA) that showed a performance considerably better than the initial GGA with an improvement rate of about 68%.

Likewise, we adopted a characterization approach based on exploratory data analysis techniques to identify the properties of difficult instances of $R||C_{max}$ and to investigate the behavior of the proposed EGGA algorithm. In this way, we proposed a set of indexes to collect important information about the instances and a collection of indexes to measure the EGGA algorithmic behavior at different stages of the search process.

The experimental results allowed identifying the $R||C_{max}$ instance characteristics that impact on their difficulty, as well as the needs of the EGGA to show a better performance. The gained knowledge was used to improve the EGGA by replacing the population initialization strategy Random min with a new one, referred to as Two-fastest-lb, that generates solutions closer to the best-known of each instance and incorporates an improvement to the mutation operator, giving rise to the Final GGA (FGGA). Finally, we performed a set of tests to analyze the efficiency and robustness of the proposed FGGA. In this order of ideas, the conducted experimental studies allowed generating the following conclusions:

1. The characterization is an useful tool to know in detail the studied problem instances and identify possible paths in the design of a solution method. The structure of the $R||C_{max}$ instances can be characterized with indexes that measure their size; the centralization, dispersion, shape, and location of processing times; particular characteristics of machines and jobs; and the existing relationships between machines and jobs. These characteristics helped in the design of the GGA presented in this work, providing information about the algorithm behavior of the designed intelligent strategies when solving $R||C_{max}$ instances with similar structures.

2. The $R||C_{max}$ instances are more difficult as they have a larger number of jobs, a high quotient of the largest and the shortest processing times and a high variability of the minimum processing times of the jobs.

3. The initial GGA, an adaptation of the GGA-CGT designed to solve the Bin Packing Problem (BPP), does not perform well on solving the $R||C_{max}$ problem, even conducting different improvements on its operators. The above reaffirms the importance of designing specific-purpose operators for the characteristics and constraints of the problem to be solved, even when the problems belong to the same class, such as BPP and $R||C_{max}$ that are grouping problems.

4. The population initialization strategy generates solutions closer to the best-known in instances with many jobs, few machines, and a low quotient q of maximum processing time by the minimum. In this sense, we observed that the designed GGA shows a better performance when it uses a population initialization strategy that generates solutions closer to the best-known. Therefore, we designed an intelligent strategy that promotes the allocation of each job to the machine that processes it fastest.

5. The crossover operator can be very disruptive when the right strategies are not used since allocating a job in the wrong machine can considerably reduce the quality of a solution. Therefore, we implement intelligent strategies to control the crossover process using criteria that promote the transmission of the most suitable machines, reducing the above disruption as far as possible. These intelligent strategies (1) order the machines from best to worst, based on their processing times C_i and the number of jobs assigned to them N_{jobs}; (2) control the gene transmission process, giving priority to the machine with the lowest C_i and, if necessary, breaking the ties randomly; (3) avoid the repeated genetic material,

removing the repeated jobs; and (4) generate only one child to evade premature convergence.

6. Unlike most of the state-of-the-art GGAs for grouping problems that use mutation operators modifying the entire groups, the search space conditions of $R||C_{max}$ instances need a mutation operator that slightly alters the solution structure by modifying only the location of certain jobs (items) in strategically selected machines (groups). Therefore, we designed a set of intelligent strategies to select the machines to mutate, pick the jobs to relocate, and rearrange the jobs when the GGA get stuck in a local optima.

7. The properties of the $R||C_{max}$ search space and the designed operators algorithmic behavior promote the generation of solutions with similar characteristics that can lead to premature convergence. To counteract this behavior, we designed a reproduction technique that uses an intelligent strategy to promote the selection of the solutions with the best characteristics for the crossover and mutation processes, discarding the solutions with similar genetic material and the same quality. Alike, this strategy promotes the replacement of solutions with similar genetic material before the worst ones. It is important to noted that the process to verify that two individuals represent the same solution has a high computational cost. Therefore, we proposed a set of simple criteria that allow identifying solutions with similar genetic material.

8. In problems with a high quotient q or correlated machines $mach\_corre$, GGA needs strategies that promote too much diversity. The index q collects information on the differences in the jobs' processing times by calculating the number of times the shortest processing time can be processed faster than the longest one. In other order of ideas, the index $mach\_corre$ measures if each machine can process all jobs in a similar time. This index is important because it allows identifying the instances with machines that process all the jobs faster than the others. Thus, a test instance with very different jobs' processing times or machines processing all the jobs faster than the others demands a search process with too much diversity. In contrast, in problems with a low quotient q or correlated jobs $jobs\_corre$, it is better to incorporate a strategy promoting diversity in a more controlled way. The index $jobs\_corre$ measures if all the machines can process each job in a similar time. This information allows identifying the instances with jobs that need a shorter processing time than the others, regardless of the machine that processes them. Therefore, a test instance with analogous jobs' processing time lengths or with machines that process every job with a similar processing time demands a search process promoting diversity in a more controlled way.

9. It is important to know in-depth the problem studied and the algorithmic behavior of the solution methods that are designed, since this information can be used to design high-performance algorithms and improve the algorithmic behavior of existing solution methods. In this sense, the approach used in this thesis project to study the GGA algorithmic behavior allowed reaching an improvement rate of around 392%, exceeding the effectiveness of the state-of-the-art solution methods by using only 10000 generations. This behavior is remarkable compared to the number of iterations required by state-of-the-art population strategies for other

grouping problems like Bin Packing [12, 402, 275]. Finally, it is important to emphasize that the designed GGA outstands in test instances with correlated jobs $JobsCorr$ and processing times in the ranges $U(1000, 1100)$ and $U(100, 120)$, where it finds better solutions than CPLEX. The instances in the $JobsCorr$ set have jobs with shorter processing times than others, regardless of the machine processing them; in this way, the jobs with the shorter processing times for all the machines can be identified easily. In other order of ideas, the sets $U(1000, 1100)$ and $U(100, 120)$ have instances with very similar jobs concerning the number of times the shortest processing time can be processed faster than the longest one. On the other hand, the instances representing the biggest challenge are in the sets $MacsCorr$, $U(1, 100)$, and $U(10, 100)$. The main particularity of the $MacsCorr$ set is that its instances have machines processing all the jobs faster than the others; therefore, the fastest machine to process all the jobs can be identified easily. On the other hand, the sets $U(1, 100)$ and $U(10, 100)$ have very different jobs for the number of times the shortest processing time can be processed faster than the longest one. In this way, the instances in the set $U(1, 100)$ have instances with job processing times that can be up to 100 times shorter than others.

The approach used to characterize the instances and solution methods, the tabular and graphical analysis models generated, and the application of the knowledge obtained respond positively to the two hypotheses raised at the beginning of this investigation (see Section 1.5). Likewise, the systematical study of the optimization process of the population initialization strategy (see Chapter 5), the crossover operator (see Chapter 6), the mutation operator (see Chapter 7), and the reproduction technique (see Chapter 8) for the $R||C_{max}$ problem, together with the characterization of the $R||C_{max}$ problem and the EGGA algorithmic behavior (Chapter 9), and the performance achieved by the proposed FGGA (Chapter 10) made possible to accomplish all the proposed objectives, as the experimental results showed that the GGA performance was considerably improved. Therefore, if we compare the initial GGA vs the FGGA with 500 generations, the FGGA presented an improvement rate of about 392%. Finally, from the contributions presented in this research, we highlight the following ones:

1. We presented the first GGA for the $R||C_{max}$ problem.
2. We presented a set of systematical studies to analyze the optimization process of each GGA component in isolation.
3. As a result, we designed intelligent strategies of specific-purpose for $R||C_{max}$, including a strategy to initialize the population, crossover and mutation operators, and a reproduction technique.
4. In this sense, we proposed the strategy to initialize the population: Two-fastest-lb that generates solutions closer to the best-known solution.
5. Moreover, we introduced the crossover operator: IE-Two machines, characterized by the way in which it sorts the machines before transmitting the genetic material, based on their processing times and the number of jobs that they have assigned.
6. Furthermore, we presented the mutation operator:2-Items Reinsertion that seeks to reduce the processing time of machines with the highest workload.

7. In addition, we designed the rearrangement strategy: Assemble based on insertion and interchange operations, which helps FGGA to get out of local optima.
8. Finally, from the systematical studies, we introduced the reproduction technique: Ranking BRW that controls selective pressure by eliminating solutions with similar characteristics.
9. The designed intelligent strategies of purpose-specific for $R||C_{max}$ were incorporated into the GGA to design the Enhanced GGA (EGGA).
10. On the other hand, we introduced and applied a collection of indexes to characterize a set of instances of the $R||C_{max}$ problem.
11. Likewise, we defined and applied a set of measures to analyze the optimization process of the EGGA and its final performance when solving $R||C_{max}$.
12. In this way, we analyzed the EGGA performance through graphs and tables that allowed obtaining explanations about its optimization process and its final performance in the solution of $R||C_{max}$ instances with different characteristics.
13. The knowledge obtained from the study of the EGGA algorithmic behavior was used to improve its performance by designing new specific-purpose operators that incorporate knowledge of the $R||C_{max}$ problem-domain. Thus, the developed operators were incorporated into the EGGA to design the Final GGA (FGGA).
14. Finally, we analyzed the FGGA robustness and efficiency through graphs and tables, which showed the usefulness of the set of systematic studies to analyze the optimization process of each GGA component in isolation and the characterization approach used to analyze the structure of the $R||C_{max}$ problem and the algorithmic behavior presented by the designed intelligent strategies when working together in FGGA. If we compare the performance of the initial GGA with an RPD of 0.0699 versus the one obtained by the FGGA of 0.0122, we can observe that the achieved improvement rate is of about 392%. Additionally, the experimental results indicate that FGGA can outstand the best results of the state-of-the-art algorithms with only 10000 generations.

11.2 Future work

From this work, we identified the following paths of work:

1. Applying the systematic experimental examination approach used in this work to analyze the algorithmic behavior of other solution methods for $R||C_{max}$ looking for extending the knowledge of the problem-domain obtained from this work.
2. Using the knowledge obtained from this work to improve the performance of state-of-the-art algorithms and to design solutions methods under other approaches, like swarm intelligence.
3. Applying the systematic experimental examination approach used in this work to other grouping problems to characterize their structure and the algorithmic behavior of the solution methods that solve them.

4. Facilitating the FGGA parameter configuration by incorporating adaptive strategies.
5. Using the FGGA proposed in this work to solve other grouping problems with similar characteristics to $R||C_{max}$.

The knowledge gained from this type of studies is useful to understand the algorithmic behavior of heuristic strategies for NP-hard problems and can be used to develop new high-performance procedures.

Bibliography

[1] Thomas Stützle. Local search algorithms for combinatorial problems-analysis, algorithms and new applications. *DISKI-Dissertationen zur Künstlichen Intelligenz, Infix, Sankt Augustin, Germany*, 1999.

[2] Michael R Garey. A guide to the theory of np-completeness. *Computers and intractability*, 1979.

[3] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[4] Ali Husseinzadeh Kashan, Ali Akbar Akbari, and Bakhtiar Ostadi. Grouping evolution strategies: An effective approach for grouping problems. *Applied Mathematical Modelling*, 39(9):2703–2720, 2015.

[5] M Quiroz-Castellanos. Caracterización del proceso de optimización de algoritmos heurísticos aplicados al problema de empacado de objetos en contenedores. *PhD in Computer Science, Instituto Tecnológico de Ciudad Madero*, 2014.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.

[7] Octavio Ramos-Figueroa, Marcela Quiroz-Castellanos, Efrén Mezura-Montes, and Oliver Schütze. Metaheuristics to solve grouping problems: A review and a case study. *Swarm and Evolutionary Computation*, page 100643, 2020.

[8] Emanuel Falkenauer. A new representation and operators for genetic algorithms applied to grouping problems. *Evolutionary computation*, 2(2):123–144, 1994.

[9] AK Bhatia and Sandip K Basu. Packing bins using multi-chromosomal genetic representation and better-fit heuristic. In *International Conference on Neural Information Processing*, pages 181–186. Springer, 2004.

[10] Tansel Dokeroglu and Ahmet Cosar. Optimization of one-dimensional bin packing problem with island parallel grouping genetic algorithms. *Computers & Industrial Engineering*, 75:176–186, 2014.

[11] Marcela Quiroz-Castellanos, Laura Cruz-Reyes, Jose Torres-Jimenez, Claudia Gómez, Héctor J Fraire Huacuja, and Adriana CF Alvim. A grouping genetic algorithm with controlled gene transmission for the bin packing problem. *Computers & Operations Research*, 55:52–64, 2015.

[12] Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2(1):5–30, 1996.

[13] Alok Singh and Ashok K Gupta. Two heuristics for the one-dimensional bin-packing problem. *OR Spectrum*, 29(4):765–781, 2007.

[14] David Wilcox, Andrew McNabb, and Kevin Seppi. Solving virtual machine packing with a reordering grouping genetic algorithm. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 362–369. IEEE, 2011.

[15] Özgür Ülker, Emin Erkan Korkmaz, and Ender Özcan. A grouping genetic algorithm using linear linkage encoding for bin packing. In *International Conference on Parallel Problem Solving from Nature*, pages 1140–1149. Springer, 2008.

[16] Tayfun Kucukyilmaz and Hakan Ezgi Kiziloz. Cooperative parallel grouping genetic algorithm for the one-dimensional bin packing problem. *Computers & Industrial Engineering*, 125:157–170, 2018.

[17] Sukru Ozer Ozcan, Tansel Dokeroglu, Ahmet Cosar, and Adnan Yazici. A novel grouping genetic algorithm for the one-dimensional bin packing problem on gpu. In *International Symposium on Computer and Information Sciences*, pages 52–60. Springer, 2016.

[18] Tabitha James, Evelyn Brown, and Cliff T Ragsdale. Grouping genetic algorithm for the blockmodel problem. *IEEE Transactions on Evolutionary Computation*, 14(1):103–111, 2009.

[19] Shyam Sundar and Alok Singh. Metaheuristic approaches for the blockmodel problem. *IEEE Systems Journal*, 9(4):1237–1247, 2014.

[20] Fereydoun Farrahi Moghaddam, Reza Farrahi Moghaddam, and Mohamed Cheriet. Carbon-aware distributed cloud: multi-level grouping genetic algorithm. *Cluster Computing*, 18(1):477–491, 2015.

[21] Michael Mutingi and Charles Mbohwa. A fuzzy grouping genetic algorithm for care assignment task. 2014.

[22] Eduardo Vila Gonçalves Filho and Alexandre José Tiberti. A group genetic algorithm for the machine cell formation problem. *International Journal of Production Economics*, 102(1):1–21, 2006.

[23] Pierre De Lit, Emanuel Falkenauer, and Alain Delchambre. Grouping genetic algorithms: an efficient method to solve the cell formation problem. *Mathematics and Computers in simulation*, 51(3-4):257–271, 2000.

[24] Evelyn C Brown and Robert T Sumichrast. Cf-gga: a grouping genetic algorithm for the cell formation problem. *International Journal of Production Research*, 39(16):3651–3669, 2001.

[25] Emmanuelle Vin, Pierre De Lit, and Alain Delchambre. A multiple-objective grouping genetic algorithm for the cell formation problem with alternative routings. *Journal of Intelligent Manufacturing*, 16(2):189–205, 2005.

[26] K Yasuda*, L Hu, and Y Yin. A grouping genetic algorithm for the multi-objective cell formation problem. *International Journal of Production Research*, 43(4):829–853, 2005.

[27] Tabitha L James, Evelyn C Brown, and Kellie B Keeling. A hybrid grouping genetic algorithm for the cell formation problem. *Computers & Operations Research*, 34(7):2059–2079, 2007.

[28] L Hu and K Yasuda. Minimising material handling cost in cell formation with alternative processing routes by grouping genetic algorithm. *International Journal of Production Research*, 44(11):2133–2167, 2006.

[29] Michael Mutingi and Godfrey C Onwubolu. Integrated cellular manufacturing system design and layout using group genetic algorithms. In *Manufacturing System*. IntechOpen, 2012.

[30] Emmanuelle Vin, Pascal Francq, and Alain Delchambre. A grouping genetic algorithm (simoggas) simultaneously to solve two grouping problems applied to the cell formation problem with alternative process plans. *Group Technology/Cellular Manufacturing (GTCM06)*, 2006.

[31] N Jawahar and R Subhaa. An adjustable grouping genetic algorithm for the design of cellular manufacturing system integrating structural and operational parameters. *Journal of Manufacturing Systems*, 44:115–142, 2017.

[32] Shyam Sundar and Alok Singh. Two grouping-based metaheuristics for clique partitioning problem. *Applied Intelligence*, 47(2):430–442, 2017.

[33] Allan Tucker, Jason Crampton, and Stephen Swift. Rgfga: An efficient representation and crossover for grouping genetic algorithms. *Evolutionary Computation*, 13(4):477–499, 2005.

[34] LE Agustı, Sancho Salcedo-Sanz, Silvia Jiménez-Fernández, Leopoldo Carro-Calvo, Javier Del Ser, José Antonio Portilla-Figueras, et al. A new grouping genetic algorithm for clustering problems. *Expert Systems with Applications*, 39(10):9695–9703, 2012.

[35] S Salcedo-Sanz, J Del Ser, and ZW Geem. An island grouping genetic algorithm for fuzzy partitioning problems. *The Scientific World Journal*, 2014, 2014.

[36] Javad Vahidi, Seyed Saeed Mirpour Marzuni, and Sara Farzai. Comparing performance of parallel grouping genetic algorithm with serial grouping genetic algorithm for clustering problems. *International Journal of Mechatronics, Electrical and Computer Technology*, 5(15):2198–2206, 2015.

[37] Sayede Houri Razavi, E Omid Mahdi Ebadati, Shahrokh Asadi, and Harleen Kaur. An efficient grouping genetic algorithm for data clustering and big data analysis. In *Computational Intelligence for Big Data Analysis*, pages 119–142. Springer, 2015.

[38] Santhosh Peddi and Alok Singh. Grouping genetic algorithm for data clustering. In *International Conference on Swarm, Evolutionary, and Memetic Computing*, pages 225–232. Springer, 2011.

[39] Emin Erkan Korkmaz, Jun Du, Reda Alhajj, and Ken Barker. Combining advantages of new chromosome representation scheme and multi-objective genetic algorithms for better clustering. *Intelligent Data Analysis*, 10(2):163–182, 2006.

[40] Peiyong Li, Chengfang Wang, and Yunsheng Mao. A hybrid grouping genetic algorithm for one-dimensional cutting stock problem. *Journal-Shanghai Jiaotong University-Chinese Edition-*, 40(6):1015, 2006.

[41] Emanuel Falkenauer. Applying genetic algorithms to real-world problems. In *Evolutionary Algorithms*, pages 65–88. Springer, 1999.

[42] Emanuel Falkenauer. Solving equal piles with the grouping genetic algorithm. In *Proceeding of the Sixth International Conference on Genetic Algorithms*, pages 492–497, 1995.

[43] Henrik Höglund. Estimating discretionary accruals using a grouping genetic algorithm. *Expert systems with applications*, 40(7):2366–2372, 2013.

[44] Michael AP Taylor. *Grouping genetic algorithm in GIS: a facility location modelling*. PhD thesis, EASTS-Eastern Asia Society for Transportation Studies, 2005.

[45] Michael Mutingi and Charles Mbohwa. *Grouping Genetic Algorithms*. Springer, 2017.

[46] Kaiji Liu, Peng Ye, Tao Hong, and Bo Li. Research of the time-dependent electric vehicle routing problem. In *Proceedings of the 2nd International Conference on Control and Computer Vision*, pages 97–101, 2019.

[47] A Aybar-Ruiz, S Jiménez-Fernández, L Cornejo-Bueno, C Casanova-Mateo, J Sanz-Justo, P Salvador-González, and S Salcedo-Sanz. A novel grouping genetic algorithm–extreme learning machine approach for global solar radiation prediction from numerical weather models inputs. *Solar Energy*, 132:129–142, 2016.

[48] Pilar García-Díaz, Isabel Sánchez-Berriel, Juan A Martínez-Rojas, and Ana M Diez-Pascual. Unsupervised feature selection algorithm for multiclass cancer classification of gene expression rna-seq data. *Genomics*, 112(2):1916–1925, 2020.

[49] James C Chen, Cheng-Chun Wu, Chia-Wen Chen, and Kou-Huang Chen. Flexible job shop scheduling with parallel machines using genetic algorithm and grouping genetic algorithm. *Expert Systems with Applications*, 39(11):10016–10021, 2012.

[50] André Rossi, Alok Singh, and Marc Sevaux. A metaheuristic for the fixed job scheduling problem under spread time constraints. *Computers & operations research*, 37(6):1045–1054, 2010.

[51] Wilhelm Erben. A grouping genetic algorithm for graph colouring and exam timetabling. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 132–156. Springer, 2000.

[52] Özgür Ülker, Ender Özcan, and Emin Erkan Korkmaz. Linear linkage encoding in grouping problems: applications on graph coloring and timetabling. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 347–363. Springer, 2006.

[53] Brahim Rekiek, Alain Delchambre, and Hussain Aziz Saleh. Handicapped person transportation: An application of the grouping genetic algorithm. *Engineering Applications of Artificial Intelligence*, 19(5):511–520, 2006.

[54] M Mutingi and Charles Mbohwa. Home healthcare worker scheduling: a group genetic algorithm approach. 2013.

[55] M Mutingi and C Mbhwa. Task assignment in home health care: A fuzzy group genetic algorithm approach. 2013.

[56] M Mutingi and C Mbohwa. Home health care staff scheduling: Effective grouping approaches. In *IAENG Transactions on Engineering Sciences-Special Issue of the International Multi-Conference of Engineers and Computer Scientists, IMECS and World Congress on Engineering, CRC Press, Taylor & Francis Group*, pages 215–224, 2014.

[57] Brahim Rekiek, Pierre De Lit, Fabrice Pellichero, Thomas L'Eglise, Patrick Fouda, Emanuel Falkenauer, and Alain Delchambre. A multiple objective grouping genetic algorithm for assembly line design. *Journal of Intelligent Manufacturing*, 12(5-6):467–485, 2001.

[58] Chang Yu Hung, Robert T Sumichrast, and Evelyn C Brown. Cpgea: a grouping genetic algorithm for material cutting plan generation. *Computers & Industrial Engineering*, 44(4):651–672, 2003.

[59] Kavita Singh and Shyam Sundar. A new hybrid genetic algorithm for the maximally diverse grouping problem. *International Journal of Machine Learning and Cybernetics*, pages 1–20, 2019.

[60] Evelyn C Brown and Mark Vroblefski. A grouping genetic algorithm for the microcell sectorization problem. *Engineering Applications of Artificial Intelligence*, 17(6):589–598, 2004.

[61] Victor B Kreng and Tseng-Pin Lee. Modular product design with grouping genetic algorithm—a case study. *Computers & Industrial Engineering*, 46(3):443–460, 2004.

[62] Michael Mutingi, Partson Dube, and Charles Mbohwa. A modular product design approach for sustainable manufacturing in a fuzzy environment. *Procedia Manufacturing*, 8:471–478, 2017.

[63] Alok Singh and Anurag Singh Baghel. A new grouping genetic algorithm for the quadratic multiple knapsack problem. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 210–218. Springer, 2007.

[64] Alex S Fukunaga. A new grouping genetic algorithm for the multiple knapsack problem. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 2225–2232. IEEE, 2008.

[65] Alok Singh and Anurag Singh Baghel. A new grouping genetic algorithm approach to the multiple traveling salesperson problem. *Soft Computing*, 13(1):95–101, 2009.

[66] Evelyn C Brown, Cliff T Ragsdale, and Arthur E Carter. Formulating the multiple traveling salesperson problem for a grouping genetic algorithm. In *IIE Annual Conference. Proceedings*, page 1. Institute of Industrial and Systems Engineers (IISE), 2004.

[67] Evelyn C Brown, Cliff T Ragsdale, and Arthur Carter. A grouping genetic algorithm for the multiple traveling salesperson problem. *International Journal of Information Technology & Decision Making*, 6(02):333–347, 2007.

[68] Dharm Raj Singh, Manoj Kumar Singh, Tarkeshwar Singh, and Rajkishore Prasad. Genetic algorithm for solving multiple traveling salesmen problem using a new crossover and population generation. *Computación y Sistemas*, 22(2), 2018.

[69] Alok Singh, Marc Sevaux, and André Rossi. A hybrid grouping genetic algorithm for multiprocessor scheduling. In *International Conference on Contemporary Computing*, pages 1–7. Springer, 2009.

[70] Jordi Balasch-Masoliver, Victor Muntés-Mulero, and Jordi Nin. Using genetic algorithms for attribute grouping in multivariate microaggregation. *Intelligent data analysis*, 18(5):819–836, 2014.

[71] Jose Alejandro Cano. Parameters for a genetic algorithm: An application for the order batching problem. *IBIMA Business Review*, 2019:802597, 2019.

[72] Giselher Pankratz. A grouping genetic algorithm for the pickup and delivery problem with time windows. *Or Spectrum*, 27(1):21–41, 2005.

[73] Mark Vroblefski and Evelyn C Brown. A grouping genetic algorithm for registration area planning. *Omega*, 34(3):220–230, 2006.

[74] Tabitha James, Mark Vroblefski, and Quinton Nottingham. A hybrid grouping genetic algorithm for the registration area planning problem. *Computer Communications*, 30(10):2180–2190, 2007.

[75] Sachchida Nand Chaurasia and Alok Singh. A hybrid evolutionary approach to the registration area planning problem. *Applied intelligence*, 41(4):1127–1149, 2014.

[76] Yuan Chen, Zhi-Ping Fan, Jian Ma, and Shuo Zeng. A hybrid grouping genetic algorithm for reviewer group construction problem. *Expert Systems with Applications*, 38(3):2401–2411, 2011.

[77] Chun-Hao Chen, Cheng-Bon Lin, and Chao-Chun Chen. Mining group stock portfolio by using grouping genetic algorithms. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 738–743. IEEE, 2015.

[78] Chun-Hao Chen and Chih-Hung Yu. A series-based group stock portfolio optimization approach using the grouping genetic algorithm with symbolic aggregate approximations. *Knowledge-Based Systems*, 125:146–163, 2017.

[79] Chun-Hao Chen, Cheng-Yu Lu, Tzung-Pei Hong, and Ja-Hwung Su. Using grouping genetic algorithm to mine diverse group stock portfolio. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 4734–4738. IEEE, 2016.

[80] Chun-Hao Chen, Yu-Hsuan Chen, Jerry Chun-Wei Lin, and Mu-En Wu. An effective approach for obtaining a group trading strategy portfolio using grouping genetic algorithm. *IEEE Access*, 7:7313–7325, 2019.

[81] Chun-Hao Chen, Cheng-Yu Lu, Tzung-Pei Hong, Jerry Chun-Wei Lin, and Matteo Gaeta. An effective approach for the diverse group stock portfolio

optimization using grouping genetic algorithm. *IEEE Access*, 7:155871–155884, 2019.

[82] Chun-Hao Chen, Wan-Yi Shen, Mu-En Wu, and Tzung-Pei Hong. A divide-and-conquer-based approach for diverse group stock portfolio optimization using island-based genetic algorithms. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 1473–1471. IEEE, 2019.

[83] Chun Hao Chen, Cheng Yu Lu, and Cheng Bon Lin. An intelligence approach for group stock portfolio optimization with a trading mechanism. *Knowledge and Information Systems*, pages 1–30, 2019.

[84] Gilberto Rivera, Luis Cisneros, Patricia Sánchez-Solís, Nelson Rangel-Valdez, and Jorge Rodas-Osollo. Genetic algorithm for scheduling optimization considering heterogeneous containers: A real-world case study. *Axioms*, 9(1):27, 2020.

[85] Lucas Cuadra, Adrián Aybar-Ruíz, MA Del Arco, Julio Navío-Marco, José Antonio Portilla-Figueras, and Sancho Salcedo-Sanz. A lamarckian hybrid grouping genetic algorithm with repair heuristics for resource assignment in wcdma networks. *Applied Soft Computing*, 43:619–632, 2016.

[86] Seyedeh Yasaman Rashida, Masoud Sabaei, Mohammad Mehdi Ebadzadeh, and Amir Masoud Rahmani. A memetic grouping genetic algorithm for cost efficient vm placement in multi-cloud environment. *Cluster Computing*, pages 1–40, 2019.

[87] Boxiong Tan, Hui Ma, and Yi Mei. A group genetic algorithm for resource allocation in container-based clouds. In *European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar)*, pages 180–196. Springer, 2020.

[88] Luis E Agustín-Blas, Sancho Salcedo-Sanz, Emilio G Ortiz-García, Antonio Portilla-Figueras, Ángel M Pérez-Bellido, and Silvia Jiménez-Fernández. Team formation based on group technology: A hybrid grouping genetic algorithm approach. *Computers & Operations Research*, 38(2):484–495, 2011.

[89] Luis E Agustín-Blas, Sancho Salcedo-Sanz, Emilio G Ortiz-García, Antonio Portilla-Figueras, and Ángel M Pérez-Bellido. A hybrid grouping genetic algorithm for assigning students to preferred laboratory groups. *Expert Systems with Applications*, 36(3):7234–7241, 2009.

[90] Yoo-Min Choi and Dong-Jin Lim. Automatic feasible transition path generation from uml state chart diagrams using grouping genetic algorithms. *Information and Software Technology*, 94:38–58, 2018.

[91] . *Model-based Test Suite Generation for Fault Localization using Search-based Mutation Testing Technique*. PhD thesis, , 2020.

[92] Luis E Agustín-Blas, Sancho Salcedo-Sanz, Pablo Vidales, G Urueta, and José Antonio Portilla-Figueras. Near optimal citywide wifi network deployment using a hybrid grouping genetic algorithm. *Expert Systems with Applications*, 38(8):9543–9556, 2011.

[93] Emanuel Falkenauer. The grouping genetic algorithms-widening the scope of the gas. *Belgian Journal of Operations Research, Statistics and Computer Science*, 33(1):2, 1992.

[94] Halil Yetgin, Kent Tsz Kan Cheung, Mohammed El-Hajjar, and Lajos Hanzo Hanzo. A survey of network lifetime maximization techniques in wireless sensor networks. *IEEE Communications Surveys & Tutorials*, 19(2):828–854, 2017.

[95] R Kamalakannan, R Sudhakara Pandian, T Sornakumar, and SS Mahapatra. An ant colony optimization algorithm for cellular manufacturing system. In *Applied Mechanics and Materials*, volume 854, pages 133–141. Trans Tech Publ, 2017.

[96] Veronique Sels, Jose Coelho, Antonio Manuel Dias, and Mario Vanhoucke. Hybrid tabu search and a truncated branch-and-bound for the unrelated parallel machine scheduling problem. *Computers & Operations Research*, 53:107–117, 2015.

[97] Dipak Laha and Jatinder ND Gupta. An improved cuckoo search algorithm for scheduling jobs on identical parallel machines. *Computers & Industrial Engineering*, 126:348–360, 2018.

[98] RM Branco and CR Rocha. Group technology: Hybrid genetic algorithm with greedy formation and a local search cluster technique in the solution of manufacturing cell formation problems. In *Book of Abstracts of the 25th International Joint Conference on Industrial Engineering and Operations*, page 21, 2019.

[99] Runwei Cheng and Mitsuo Gen. Parallel machine scheduling problems using memetic algorithms. In *1996 IEEE International Conference on Systems, Man and Cybernetics. Information Intelligence and Systems (Cat. No. 96CH35929)*, volume 4, pages 2665–2670. IEEE, 1996.

[100] Claus de Castro Aranha and Hitoshi Iba. Using memetic algorithms to improve portfolio performance in static and dynamic trading scenarios. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 1427–1434. ACM, 2009.

[101] Christian Blum, Jakob Puchinger, Günther R Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011.

[102] Mirsad Buljubasic. *Efficient local search for several combinatorial optimization problems*. Theses, Université Montpellier, November 2015.

[103] Marco Chiarandini, Irina Dumitrescu, and Thomas Stützle. Stochastic local search algorithms for the graph colouring problem. *Computer & Information Science Series, Chapman & Hall, CRC*, 2018.

[104] Haroldo G Santos, Túlio AM Toffolo, Cristiano LTF Silva, and Greet Vanden Berghe. Analysis of stochastic local search methods for the unrelated parallel machine scheduling problem. *International Transactions in Operational Research*, 26(2):707–724, 2019.

[105] Antonio Martinez-Sykora, Ramón Alvarez-Valdés, Julia A Bennell, R Ruiz, and José Manuel Tamarit. Matheuristics for the irregular bin packing problem with free rotations. *European Journal of Operational Research*, 258(2):440–455, 2017.

[106] Zeping Pei, Zhuan Wang, and Yiwen Yang. Research of order batching variable neighborhood search algorithm based on saving mileage. In *3rd International*

Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019). Atlantis Press, 2019.

[107] Thomas Kämpke. Simulated annealing: use of a new tool in bin packing. *Annals of Operations Research*, 16(1):327–332, 1988.

[108] Daniel Schermer, Mahdi Moeini, and Oliver Wendt. A hybrid vns/tabu search algorithm for solving the vehicle routing problem with drones and en route operations. *Computers & Operations Research*, 109:134–158, 2019.

[109] Brototi Mondal, Kousik Dasgupta, and Paramartha Dutta. Load balancing in cloud computing using stochastic hill climbing-a soft computing approach. *Procedia Technology*, 4:783–789, 2012.

[110] Biao Yuan, Chaoyong Zhang, and Xinyu Shao. A late acceptance hill-climbing algorithm for balancing two-sided assembly lines with multiple constraints. *Journal of Intelligent Manufacturing*, 26(1):159–168, 2015.

[111] F Yu Vincent and Shih-Wei Lin. Multi-start simulated annealing heuristic for the location routing problem with simultaneous pickup and delivery. *Applied soft computing*, 24:284–290, 2014.

[112] Shinji Sakamoto, Elis Kulla, Tetsuya Oda, Makoto Ikeda, Leonard Barolli, and Fatos Xhafa. A comparison study of hill climbing, simulated annealing and genetic algorithm for node placement problem in wmns. *Journal of High Speed Networks*, 20(1):55–66, 2014.

[113] M Emin Aydin and Terence C Fogarty. A simulated annealing algorithm for multi-agent systems: a job-shop scheduling application. *Journal of intelligent manufacturing*, 15(6):805–814, 2004.

[114] Amanda Hiley and Bryant A Julstrom. The quadratic multiple knapsack problem and three heuristic approaches to it. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 547–552. ACM, 2006.

[115] Mhand Hifi. Dynamic programming and hill-climbing techniques for constrained two-dimensional cutting stock problems. *Journal of combinatorial optimization*, 8(1):65–84, 2004.

[116] Marina Yusoff and Nurhikmah Roslan. Evaluation of genetic algorithm and hybrid genetic algorithm-hill climbing with elitist for lecturer university timetabling problem. In *International Conference on Swarm Intelligence*, pages 363–373. Springer, 2019.

[117] Laith Mohammad Abualigah, Ahamad Tajudin Khader, Mohammed Azmi Al-Betar, Zaid Abdi Alkareem Alyasseri, Osama Ahmad Alomari, and Essam Said Hanandeh. Feature selection with β-hill climbing search for text clustering application. In *2017 Palestinian International Conference on Information and Communication Technology (PICICT)*, pages 22–27. IEEE, 2017.

[118] Fan Wang and Zhou Xu. Metaheuristics for robust graph coloring. *Journal of Heuristics*, 19(4):529–548, 2013.

[119] Sebastian Henn and Verena Schmid. Metaheuristics for order batching and sequencing in manual order picking systems. *Computers & Industrial Engineering*, 66(2):338–351, 2013.

[120] Edilson Reis Rodrigues Kato, Gabriel Diego de Aguiar Aranha, and Roberto Hideaki Tsunaki. A new approach to solve the flexible job shop problem based on a hybrid particle swarm optimization and random-restart hill climbing. *Computers & Industrial Engineering*, 125:178–189, 2018.

[121] Luiz FO Moura Santos, Renan Sallai Iwayama, Luísa Brandão Cavalcanti, Leandro Maciel Turi, Fabio Emanuel de Souza Morais, Gabriel Mormilho, and Claudio B Cunha. A variable neighborhood search algorithm for the bin packing problem with compatible categories. *Expert Systems with Applications*, 124:209–225, 2019.

[122] I Davydov and Yury Kochetov. Vns-based heuristic with an exponential neighborhood for the server load balancing problem. *Electronic Notes in Discrete Mathematics*, 47:53–60, 2015.

[123] Masood Fathi, Amir Nourmohammadi, Amos HC Ng, Anna Syberfeldt, and Hamidreza Eskandari. An improved genetic algorithm with variable neighborhood search to solve the assembly line balancing problem. *Engineering Computations*, 2019.

[124] Jiawen Lu and Ling Wang. A bi-strategy based optimization algorithm for the dynamic capacitated electric vehicle routing problem. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 646–653. IEEE, 2019.

[125] Ivan C Martins, Rian GS Pinheiro, Fábio Protti, and Luiz S Ochi. A hybrid iterated local search and variable neighborhood descent heuristic applied to the cell formation problem. *Expert Systems with Applications*, 42(22):8947–8955, 2015.

[126] Yongzhen Wang, Yan Chen, and Yan Lin. Memetic algorithm based on sequential variable neighborhood descent for the minmax multiple traveling salesman problem. *Computers & Industrial Engineering*, 106:105–122, 2017.

[127] Ivan Davydov, Yury Kochetov, and Stephan Dempe. Local search approach for the competitive facility location problem in mobile networks. *International Journal of Artificial Intelligence*, 16(1):130–143, 2018.

[128] Fuqing Zhao, Shuo Qin, Yi Zhang, Weimin Ma, Chuck Zhang, and Houbin Song. A hybrid biogeography-based optimization with variable neighborhood search mechanism for no-wait flow shop scheduling problem. *Expert Systems with Applications*, 126:321–339, 2019.

[129] France Roanne. A variable neighborhood search with integer programming for the zero-one multiple-choice knapsack problem with setup. *Variable Neighborhood Search*, page 152, 2019.

[130] Frederico Dusberger and Günther R Raidl. Solving the 3-staged 2-dimensional cutting stock problem by dynamic programming and variable neighborhood search. *Electronic Notes in Discrete Mathematics*, 47:133–140, 2015.

[131] Rafidah Abdul Aziz, Masri Ayob, Zalinda Othman, Zulkifli Ahmad, and Nasser R Sabar. An adaptive guided variable neighborhood search based on honey-bee mating optimization algorithm for the course timetabling problem. *Soft Computing*, 21(22):6755–6765, 2017.

[132] Jack Brimberg, Nenad Mladenović, Raca Todosijević, and Dragan Urošević. Solving the capacitated clustering problem with variable neighborhood search. *Annals of Operations Research*, 272(1-2):289–321, 2019.

[133] Gintaras Palubeckis, Eimutis Karčiauskas, and Aleksas Riškus. Comparative performance of three metaheuristic approaches for the maximally diverse grouping problem. *Information Technology and Control*, 40(4):277–285, 2011.

[134] Dragan Matic, Jozef Kratica, and Vladimir Filipovic. Variable neighborhood search for solving bandwidth coloring problem. *Computer Science and Information Systems*, 14(2):309–327, 2015.

[135] Yin-Yann Chen, Chen-Yang Cheng, Li-Chih Wang, and Tzu-Li Chen. A hybrid approach based on the variable neighborhood search and particle swarm optimization for parallel machine scheduling problems—a case study for solar cell industry. *International Journal of Production Economics*, 141(1):66–78, 2013.

[136] Sana Frifita, Malek Masmoudi, and Jalel Euchi. General variable neighborhood search for home healthcare routing and scheduling problem with time windows and synchronized visits. *Electronic Notes in Discrete Mathematics*, 58:63–70, 2017.

[137] Xu Yingzhuo and Geng Qing Yang. Research on network load balancing method based on simulated annealing algorithm and genetic algorithm. In *Journal of Physics: Conference Series*, volume 1237, page 022137. IOP Publishing, 2019.

[138] M Yang, L Ba, Y Liu, HY Zheng, JT Yan, XQ Gao, and JM Xiao. An improved genetic simulated annealing algorithm for stochastic two-sided assembly line balancing problem. *Int simul model*, 18:175–186, 2019.

[139] Kenan Karagul, Yusuf Sahin, Erdal Aydemir, and Aykut Oral. A simulated annealing algorithm based solution method for a green vehicle routing problem with fuel consumption. In *Lean and green supply chain management*, pages 161–187. Springer, 2019.

[140] R Kamalakannan, R Sudhakara Pandian, and P Sivakumar. A simulated annealing for the cell formation problem with ratio level data. *International Journal of Enterprise Network Management*, 10(1):78–90, 2019.

[141] Chi Hwa Song, Kyunghee Lee, and Won Don Lee. Extended simulated annealing for augmented tsp and multi-salesmen tsp. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 3, pages 2340–2343. IEEE, 2003.

[142] Kamyla Maria Ferreira and Thiago Alves de Queiroz. Two effective simulated annealing algorithms for the location-routing problem. *Applied Soft Computing*, 70:389–422, 2018.

[143] Fernando Garza-Santisteban, Roberto Sánchez-Pámanes, Luis Antonio Puente-Rodríguez, Ivan Amaya, José Carlos Ortiz-Bayliss, Santiago Conant-Pablos, and Hugo Terashima-Marín. A simulated annealing hyper-heuristic for job shop scheduling problems. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 57–64. IEEE, 2019.

[144] Adil Baykasoglu, Turkay Dereli, and Sena Das. Project team selection using fuzzy optimization approach. *Cybernetics and Systems: An International Journal*, 38(2):155–185, 2007.

[145] Ivan Adrian Lopez Sanchez, Jaime Mora Vargas, Cipriano A Santos, Miguel Gonzalez Mendoza, and Cesar J Montiel Moctezuma. Solving binary cutting stock with matheuristics using particle swarm optimization and simulated annealing. *Soft Computing*, 22(18):6111–6119, 2018.

[146] Nuno Leite, Fernando Melício, and Agostinho C Rosa. A fast simulated annealing algorithm for the examination timetabling problem. *Expert Systems with Applications*, 122:137–151, 2019.

[147] Sattar Seifollahi, Adil Bagirov, Ehsan Zare Borzeshi, and Massimo Piccardi. A simulated annealing-based maximum-margin clustering algorithm. *Computational Intelligence*, 35(1):23–41, 2019.

[148] Alper Kose, Berke Aral Sonmez, and Metin Balaban. Simulated annealing algorithm for graph coloring. *arXiv preprint arXiv:1712.00709*, 2017.

[149] Shih-Wei Lin and Kuo-Ching Ying. A multi-point simulated annealing heuristic for solving multiple objective unrelated parallel machine scheduling problems. *International Journal of Production Research*, 53(4):1065–1076, 2015.

[150] Eric H Grosse, Christoph H Glock, Rafael Ballester-Ripoll, et al. A simulated annealing approach for the joint order batching and order picker routing problem with weight restrictions. *International Journal of Operations and Quantitative Management*, 20(2):65–83, 2014.

[151] Amir Mohammad Fathollahi-Fard, Kannan Govindan, Mostafa Hajiaghaei-Keshteli, and Abbas Ahmadi. A green home health care supply chain: New modified simulated annealing algorithms. *Journal of Cleaner Production*, 240:118200, 2019.

[152] Yves Crama and Michaël Schyns. Simulated annealing for complex portfolio selection problems. *European Journal of operational research*, 150(3):546–571, 2003.

[153] Joaquim L Viegas, Susana M Vieira, Elsa MP Henriques, and Joao MC Sousa. A tabu search algorithm for the 3d bin packing problem in the steel industry. In *CONTROLO'2014–Proceedings of the 11th Portuguese Conference on Automatic Control*, pages 355–364. Springer, 2015.

[154] Nadim Téllez, Miguel Jimeno, Augusto Salazar, and E Nino-Ruiz. A tabu search method for load balancing in fog computing. *Int. Artif. Intell*, 16(2):1–31, 2018.

[155] Kadir Buyukozkan, Ibrahim Kucukkoc, Sule Itir Satoglu, and David Z Zhang. Lexicographic bottleneck mixed-model assembly line balancing problem: artificial bee colony and tabu search approaches with optimised parameters. *Expert Systems with Applications*, 50:151–166, 2016.

[156] Farhad Ghassemi Tari and Khatereh Ahadi. Cellular layout design using tabu search, a case study. *RAIRO-Operations Research*, 53(5):1475–1488, 2019.

[157] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, 2006.

[158] Mauricio Romero Montoya, Rogelio González Velázquez, Martín Estrada Analco, José Luis Martínez Flores, and María Beatriz Bernábe Loranca. Solution search for the capacitated p-median problem using tabu search. *International Journal of Combinatorial Optimization Problems and Informatics*, 10(2):17–25, 2019.

[159] Jun-Qiang Li, Peiyong Duan, Jinde Cao, Xiao-Ping Lin, and Yu-Yan Han. A hybrid pareto-based tabu search for the distributed flexible job shop scheduling problem with e/t criteria. *IEEE Access*, 6:58883–58897, 2018.

[160] Roberto Aringhieri. Composing medical crews with equity and efficiency. *Central European Journal of Operations Research*, 17(3):343–357, 2009.

[161] Jin Qin, Xianhao Xu, Qinghua Wu, and TCE Cheng. Hybridization of tabu search with feasible and infeasible local searches for the quadratic multiple knapsack problem. *Computers & Operations Research*, 66:199–214, 2016.

[162] Meghdad HMA Jahromi, Reza Tavakkoli-Moghaddam, Ahmad Makui, and Abbas Shamsi. Solving an one-dimensional cutting stock problem by simulated annealing and tabu search. *Journal of Industrial Engineering International*, 8(1):24, 2012.

[163] Paula Amaral and Tiago Cardal Pais. Compromise ratio with weighting functions in a tabu search multi-criteria approach to examination timetabling. *Computers & Operations Research*, 72:160–174, 2016.

[164] Ali Falah Yaqoob and Basad Al-Sarray. Finding best clustering for big networks with minimum objective function by using probabilistic tabu search. *Iraqi Journal of Science*, 60(8):1837–1845, 2019.

[165] Abraham Duarte and Rafael Martí. Tabu search and grasp for the maximum diversity problem. *European Journal of Operational Research*, 178(1):71–84, 2007.

[166] Sebastian Henn and Gerhard Wäscher. Tabu search heuristics for the order batching problem in manual order picking systems. *European Journal of Operational Research*, 222(3):484–494, 2012.

[167] Zhuo Yihe, Liu Ran, and Hua Yikang. Tabu search algorithm for periodic home health care problem. *China Sciencepaper*, (14):22, 2015.

[168] Majid M Aldaihani and Talla M Al-Deehani. Mathematical models and a tabu search for the portfolio management problem in the kuwait stock exchange. *International Journal of Operational Research*, 7(4):445–462, 2010.

[169] Chong Peng, Guanglin Wu, T Warren Liao, and Hedong Wang. Research on multi-agent genetic algorithm based on tabu search for the job shop scheduling problem. *PloS one*, 14(9), 2019.

[170] ore Stakic, Ana Anokic, and Raka Jovanovic. Comparison of different grasp algorithms for the heterogeneous vector bin packing problem. In *2019 China-Qatar International Workshop on Artificial Intelligence and Applications to Intelligent Manufacturing (AIAIM)*, pages 63–70. IEEE, 2019.

[171] Sisca Octarina, Sugandi Yahdin, and Belly Wardhani. Implementasi algoritma greedy randomized adaptive search procedure (grasp) dan formulasi model dotted

board pada penyelesaian cutting stock problem bentuk irregular. In *Annual Research Seminar (ARS)*, volume 4, pages 228–233, 2019.

[172] Temel Öncan. Milp formulations and an iterated local search algorithm with tabu thresholding for the order batching problem. *European Journal of Operational Research*, 243(1):142–155, 2015.

[173] Mariana de Siqueira Guersola and Maria Teresinha Arns Steiner. Iterated local search adapted to clustering and routing problems. In *2015 Latin America Congress on Computational Intelligence (LA-CCI)*, pages 1–6. IEEE, 2015.

[174] Eduardo Raul Hruschka, Ricardo JGB Campello, Alex A Freitas, et al. A survey of evolutionary algorithms for clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 39(2):133–155, 2009.

[175] Soukaina Laabadi, Mohamed Naimi, Hassan El Amri, and Boujemâa Achchab. A crow search-based genetic algorithm for solving two-dimensional bin packing problem. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 203–215. Springer, 2019.

[176] Omid Homaee, Arsalan Najafi, Mohammad Dehghanian, Mehdi Attar, and Hamid Falaghi. A practical approach for distribution network load balancing by optimal re-phasing of single phase customers using discrete genetic algorithm. *International Transactions on Electrical Energy Systems*, 29(5):e2834, 2019.

[177] Han-ye Zhang. An immune genetic algorithm for simple assembly line balancing problem of type 1. *Assembly Automation*, 39(1):113–123, 2019.

[178] Ali Baniamerian, Mahdi Bashiri, and Reza Tavakkoli-Moghaddam. Modified variable neighborhood search and genetic algorithm for profitable heterogeneous vehicle routing problem with cross-docking. *Applied Soft Computing*, 75:441–460, 2019.

[179] Yu Zhu and Lin Wu. Structure study of multiple traveling salesman problem using genetic algorithm. In *2019 34rd Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pages 323–328. IEEE, 2019.

[180] Fabrício Lacerda Biajoli, Antonio Augusto Chaves, and Luiz Antonio Nogueira Lorena. A biased random-key genetic algorithm for the two-stage capacitated facility location problem. *Expert Systems with Applications*, 115:418–426, 2019.

[181] Ali Asghar Rahmani Hosseinabadi, Javad Vahidi, Behzad Saemi, Arun Kumar Sangaiah, and Mohamed Elhoseny. Extended genetic algorithm for solving open-shop scheduling problem. *Soft computing*, 23(13):5099–5116, 2019.

[182] Hyeongon Wi, Seungjin Oh, Jungtae Mun, and Mooyoung Jung. A team formation model based on knowledge and collaboration. *Expert Systems with Applications*, 36(5):9121–9134, 2009.

[183] Roozbeh Sanaei, Kevin Otto, Kristin Wood, Katja Hölttä-Otto, et al. A rapid algorithm for multi-objective pareto optimization of modular architecture. In *DS 87-4 Proceedings of the 21st International Conference on Engineering Design (ICED 17) Vol 4: Design Methods and Tools, Vancouver, Canada, 21-25.08. 2017*, pages 169–178, 2017.

[184] Ebaa Fayyoumi and Omar Nofal. Applying genetic algorithms on multi-level micro-aggregation techniques for secure statistical databases. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–6. IEEE, 2018.

[185] Ali Nadi Ünal. A genetic algorithm for the multiple knapsack problem in dynamic environment. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 2, 2013.

[186] Anton Orlov, Vladimir Kureichik, and Alexander Glushchenko. Hybrid genetic algorithm for cutting stock and packaging problems. In *2016 IEEE East-West Design & Test Symposium (EWDTS)*, pages 1–4. IEEE, 2016.

[187] MA El-Shorbagy, AY Ayoub, AA Mousa, and IM El-Desoky. An enhanced genetic algorithm with new mutation for cluster analysis. *Computational Statistics*, pages 1–38, 2019.

[188] Jing Luan, Zhong Yao, Futao Zhao, and Xin Song. A novel method to solve supplier selection problem: Hybrid algorithm of genetic algorithm and ant colony optimization. *Mathematics and Computers in Simulation*, 156:294–309, 2019.

[189] Said Labed, KOUT Akram, and Salim Chikhi. Solving the graph b-coloring problem with hybrid genetic algorithm. In *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–7. IEEE, 2018.

[190] Muflih Hafidz Danurhadi, Dida Diah Damayanti, and Widia Juliani. Identical parallel machine scheduling using genetic algorithm to minimize total tardiness for cnc 4 axis in pt dirgantara indonesia (persero). *eProceedings of Engineering*, 6(2), 2019.

[191] Chih Ming Hsu, Kai Ying Chen, and Mu Chen Chen. Batching orders in warehouses by minimizing travel distance with genetic algorithms. *Computers in Industry*, 56(2):169–178, 2005.

[192] Rahma Borchani, Malek Masmoudi, and Bassem Jarboui. Hybrid genetic algorithm for home healthcare routing and scheduling problem. In *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 1900–1904. IEEE, 2019.

[193] R Yusuf, BD Handari, and GF Hertono. Implementation of agglomerative clustering and genetic algorithm on stock portfolio optimization with possibilistic constraints. In *AIP Conference Proceedings*, volume 2168, page 020028. AIP Publishing, 2019.

[194] A Yoosefelahi, M Aminnayeri, H Mosadegh, and H Davari Ardakani. Type ii robotic assembly line balancing problem: An evolution strategies algorithm for a multi-objective model. *Journal of Manufacturing Systems*, 31(2):139–151, 2012.

[195] DAVID Mester. An evolutionary strategies algorithm for large scale vehicle routing problem with capacitate and time windows restrictions. In *Proceedings of the Conference on Mathematical and Population Genetics, University of Haifa, Israel*, 2002.

[196] Adam Stawowy. Evolutionary strategy for manufacturing cell design. *Omega*, 34(1):1–18, 2006.

[197] Alejandro Teran-Somohano and Alice E Smith. Locating multiple capacitated semi-obnoxious facilities using evolutionary strategies. *Computers & Industrial Engineering*, 133:303–316, 2019.

[198] Ramiro Varela, Alberto Gomez, Camino R Vela, Jorge Puente, and Cesar Alonso. Heuristic generation of the initial population in solving job shop problems by evolutionary strategies. In *International Work-Conference on Artificial Neural Networks*, pages 690–699. Springer, 1999.

[199] Han Wang, Zhilei Ren, Xiaochen Li, and He Jiang. Solving team making problem for crowdsourcing with evolutionary strategy. In *2018 5th International Conference on Dependable Systems and Their Applications (DSA)*, pages 65–74. IEEE, 2018.

[200] Samir Ribić and Samim Konjicija. Evolution strategy to make an objective function in two-phase ilp timetabling. In *2011 19thTelecommunications Forum (TELFOR) Proceedings of Papers*, pages 1486–1489. IEEE, 2011.

[201] C-Y Lee and EK Antonsson. Dynamic partitional clustering using evolution strategies. In *2000 26th Annual Conference of the IEEE Industrial Electronics Society. IECON 2000. 2000 IEEE International Conference on Industrial Electronics, Control and Instrumentation. 21st Century Technologies*, volume 4, pages 2716–2721. IEEE, 2000.

[202] Alan Robert Resende de Freitas, Frederico Gadelha Guimarães, Rodrigo César Pedrosa Silva, and Marcone Jamilson Freitas Souza. Memetic self-adaptive evolution strategies applied to the maximum diversity problem. *Optimization Letters*, 8(2):705–714, 2014.

[203] Chiuh-Cheng Chyu and Wei-Shung Chang. A competitive evolution strategy memetic algorithm for unrelated parallel machine scheduling to minimize total weighted tardiness and flow time. In *The 40th International Conference on Computers & Indutrial Engineering*, pages 1–6. IEEE, 2010.

[204] Piotr Lipinski, Katarzyna Winczura, and Joanna Wojcik. Building risk-optimal portfolio using evolutionary strategies. In *Workshops on Applications of Evolutionary Computation*, pages 208–217. Springer, 2007.

[205] László Kota and Karoly Jarmai. Mathematical modeling of multiple tour multiple traveling salesman problem using evolutionary programming. *Applied Mathematical Modelling*, 39(12):3410–3433, 2015.

[206] Raymond Chiong, Yang Yaw Chang, Pui Chang Chai, and Ai Leong Wong. A selective mutation based evolutionary programming for solving cutting stock problem without contiguity. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 1671–1677. IEEE, 2008.

[207] Gursel A Suer. Evolutionary programming for designing manufacturing cells. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*, pages 379–384. IEEE, 1997.

[208] Ling Wang and Da-Zhong Zheng. A modified evolutionary programming for flow shop scheduling. *The International Journal of Advanced Manufacturing Technology*, 22(7-8):522–527, 2003.

[209] Ai-Qing Yu and Xing-Sheng Gu. Hybrid quantum-inspired evolutionary programming for identical parallel machines scheduling. *Control and Decision*, 26(10):1473–1478, 2011.

[210] Qing Tan, Qing He, Weizhong Zhao, Zhongzhi Shi, and E Stanley Lee. An improved fcmbp fuzzy clustering method based on evolutionary programming. *Computers & Mathematics with Applications*, 61(4):1129–1144, 2011.

[211] Kevin Sim and Emma Hart. Generating single and multiple cooperative heuristics for the one dimensional bin packing problem using a single node genetic programming island model. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1549–1556. ACM, 2013.

[212] Adil Baykasoglu and Lale Ozbakir. Discovering task assignment rules for assembly line balancing via genetic programming. *The International Journal of Advanced Manufacturing Technology*, 76(1-4):417–434, 2015.

[213] Liang Feng, Yew-Soon Ong, Caishun Chen, and Xianshun Chen. Conceptual modeling of evolvable local searches in memetic algorithms using linear genetic programming: a case study on capacitated vehicle routing problem. *Soft Computing*, 20(9):3745–3769, 2016.

[214] Christos Dimopoulos. A genetic programming methodology for the solution of the multiobjective cell-formation problem. In *Proceedings of the 8th Joint Conference in Information Systems (JCIS 2005)*, pages 1487–1494, 2005.

[215] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Genetic programming for job shop scheduling. In *Evolutionary and Swarm Intelligence Algorithms*, pages 143–167. Springer, 2019.

[216] John H Drake, Matthew Hyde, Khaled Ibrahim, and Ender Ozcan. A genetic programming hyper-heuristic for the multidimensional knapsack problem. *Kybernetes*, 43(9/10):1500–1511, 2014.

[217] Rushil Raghavjee and Nelishia Pillay. A comparison of genetic algorithms and genetic programming in solving the school timetabling problem. In *2012 Fourth World Congress on Nature and Biologically Inspired Computing (NaBIC)*, pages 98–103. IEEE, 2012.

[218] Andrew Lensen, Bing Xue, and Mengjie Zhang. Genetic programming for evolving similarity functions for clustering: Representations and analysis. *Evolutionary computation*, pages 1–29, 2019.

[219] Alireza Fallahpour, Ezutah Udoncy Olugu, Siti Nurmaya Musa, Dariush Khezrimotlagh, and Kuan Yew Wong. An integrated model for green supplier selection under fuzzy environment: application of data envelopment analysis and genetic programming approach. *Neural Computing and Applications*, 27(3):707–725, 2016.

[220] Paresh Tolay and Rajeev Kumar. Evolution of hyperheuristics for the biobjective graph coloring problem using multiobjective genetic programming. In *Proceedings*

of the 11th Annual conference on Genetic and evolutionary computation, pages 1939–1940. ACM, 2009.

[221] Marko Durasević, Domagoj Jakobović, and Karlo Knežević. Adaptive scheduling on unrelated machines with genetic programming. *Applied Soft Computing*, 48:419–430, 2016.

[222] Liad Wagman. Stock portfolio evaluation: An application of genetic-programming-based technical analysis. *Genetic Algorithms and Genetic Programming at Stanford*, 2003:213–220, 2003.

[223] Juan Carlos Gomez and Hugo Terashima-Marín. Evolutionary hyper-heuristics for tackling bi-objective 2d bin packing problems. *Genetic Programming and Evolvable Machines*, 19(1-2):151–181, 2018.

[224] Poontana Sresracoo, Nuchsara Kriengkorakot, Preecha Kriengkorakot, and Krit Chantarasamai. U-shaped assembly line balancing by using differential evolution algorithm. *Mathematical and Computational Applications*, 23(4):79, 2018.

[225] Siwaporn Kunnapapdeelert and Ratchaphong Klinsrisuk. Determination of green vehicle routing problem via differential evolution. *International Journal of Logistics Systems and Management*, 34(3):395–410, 2019.

[226] Jin Kiat Chong and Xin Qiu. An opposition-based self-adaptive differential evolution with decomposition for solving the multiobjective multiple salesman problem. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 4096–4103. IEEE, 2016.

[227] Rui Chi, Yixin Su, Zhijian Qu, and Xuexin Chi. A hybridization of cuckoo search and differential evolution for the logistics distribution center location problem. *Mathematical Problems in Engineering*, 2019, 2019.

[228] Xiuli Wu, Xiajing Liu, and Ning Zhao. An improved differential evolution algorithm for solving a distributed assembly flexible job shop scheduling problem. *Memetic Computing*, 11(4):335–355, 2019.

[229] Deng Libao, Wang Sha, Jin Chengyu, and Hu Cong. A hybrid mutation scheme-based discrete differential evolution algorithm for multidimensional knapsack problem. In *2016 Sixth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*, pages 1009–1014. IEEE, 2016.

[230] Khalid Shaker, Salwani Abdullah, and Arwa Hatem. A differential evolution algorithm for the university course timetabling problem. In *2012 4th Conference on Data Mining and Optimization (DMO)*, pages 99–102. IEEE, 2012.

[231] Mohammed Alswaitti, Mohanad Albughdadi, and Nor Ashidi Mat Isa. Variance-based differential evolution algorithm with an optional crossover for data clustering. *Applied Soft Computing*, 80:1–17, 2019.

[232] Sunil Kumar Jauhar and Millie Pant. Sustainable supplier selection: a new differential evolution strategy with automotive industry application. In *Recent developments and new direction in soft-computing foundations and applications*, pages 353–371. Springer, 2016.

[233] Iztok Fister and Janez Brest. Using differential evolution for the graph coloring. In *2011 IEEE Symposium on Differential Evolution (SDE)*, pages 1–7. IEEE, 2011.

[234] Xueqi Wu and Ada Che. A memetic differential evolution algorithm for energy-efficient parallel machine scheduling. *Omega*, 82:155–165, 2019.

[235] AA Adebiyi and CK Ayo. Portfolio selection problem using generalized differential evolution 3. *Applied Mathematical Sciences*, 9(42):2069–2082, 2015.

[236] Kaustabha Ray, Sunanda Bose, and Nandini Mukherjee. A load balancing approach to resource provisioning in cloud infrastructure with a grouping genetic algorithm. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*, pages 1–6. IEEE, 2018.

[237] Murat Şahin and Talip Kellegöz. An efficient grouping genetic algorithm for u-shaped assembly line balancing problems with maximizing production rate. *Memetic Computing*, 9(3):213–229, 2017.

[238] Abdeljawed Sadok, Jacques Teghem, and Habib Chabchoub. A hybrid grouping genetic algorithm for the inventory routing problem with multi-tours of the vehicle. *International Journal of Combinatorial Optimization Problems and Informatics*, 1(2):42–61, 2010.

[239] Emmanuelle Vin and Alain Delchambre. Generalized cell formation: iterative versus simultaneous resolution with grouping genetic algorithm. *Journal of intelligent manufacturing*, 25(5):1113–1124, 2014.

[240] Ladda Pitaksringkarn and Michael AP Taylor. Grouping genetic alogirhtm in gis: A facility location modelling. *Journal of the Eastern Asia Society for Transportation Studies*, 6:2908–2920, 2005.

[241] Michael Mutingi and Charles Mbohwa. Modeling modular design for sustainable manufacturing: A fuzzy grouping genetic algorithm approach. In *Grouping Genetic Algorithms*, pages 199–211. Springer, 2017.

[242] Sami Khuri, Tim Walters, and Yanti Sugono. A grouping genetic algorithm for coloring the edges of graphs. In *SAC (1)*, pages 422–427. Citeseer, 2000.

[243] Michael Mutingi and Charles Mbohwa. Modeling supplier selection using multi-criterion fuzzy grouping genetic algorithm. In *Grouping Genetic Algorithms*, pages 213–228. Springer, 2017.

[244] Michael Mutingi and Charles Mbohwa. Optimizing order batching in order picking systems: Hybrid grouping genetic algorithm. In *Grouping Genetic Algorithms*, pages 121–140. Springer, 2017.

[245] Mazyar Ghadiri Nejad, Ali Husseinzadeh Kashan, and Seyed Mahdi Shavarani. A novel competitive hybrid approach based on grouping evolution strategy algorithm for solving u-shaped assembly line balancing problems. *Production Engineering*, 12(5):555–566, 2018.

[246] Ali Husseinzadeh Kashan, Babak Rezaee, and Somayyeh Karimiyan. An efficient approach for unsupervised fuzzy clustering based on grouping evolution strategies. *Pattern Recognition*, 46(5):1240–1254, 2013.

[247] Ali Husseinzadeh Kashan, Marziehsadat Keshmiry, Jalil Heidary Dahooie, and Amin Abbasi-Pooya. A simple yet effective grouping evolutionary strategy (ges) algorithm for scheduling parallel machines. *Neural Computing and Applications*, 30(6):1925–1938, 2018.

[248] Hong-Fang Yan, Ci-Yun Cai, De-Huai Liu, and Min-Xia Zhang. Water wave optimization for the multidimensional knapsack problem. In *International Conference on Intelligent Computing*, pages 688–699. Springer, 2019.

[249] Fuqing Zhao, Huan Liu, Yi Zhang, Weimin Ma, and Chuck Zhang. A discrete water wave optimization algorithm for no-wait flow shop scheduling problem. *Expert Systems with Applications*, 91:347–363, 2018.

[250] Simone A Ludwig and Azin Moallem. Swarm intelligence approaches for grid load balancing. *Journal of Grid Computing*, 9(3):279–301, 2011.

[251] Bassem Jarboui, Saber Ibrahim, Patrick Siarry, and Abdelwaheb Rebai. A combinatorial particle swarm optimisation for solving permutation flowshop problems. *Computers & Industrial Engineering*, 54(3):526–538, 2008.

[252] Absalom E Ezugwu and Francis Akutsah. An improved firefly algorithm for the unrelated parallel machines scheduling problem with sequence-dependent setup times. *IEEE Access*, 6:54459–54478, 2018.

[253] Sara Tabaghchi Milan, Lila Rajabion, Hamideh Ranjbar, and Nima Jafari Navimipoir. Nature inspired meta-heuristic algorithms for solving the load-balancing problem in cloud environments. *Computers & Operations Research*, 2019.

[254] Georgi Evtimov and Stefka Fidanova. Ant colony optimization algorithm for 1d cutting stock problem. In *Advanced Computing in Industrial Mathematics*, pages 25–31. Springer, 2018.

[255] Thatchai Thepphakorn, Pupong Pongcharoen, and Srisatja Vitayasak. A new multiple objective cuckoo search for university course timetabling problem. In *International Workshop on Multi-disciplinary Trends in Artificial Intelligence*, pages 196–207. Springer, 2016.

[256] Ming Hao Xue, Tie Zhu Wang, and Sheng Mao. Double evolutsional artificial bee colony algorithm for multiple traveling salesman problem. In *MATEC Web of Conferences*, volume 44, page 02025. EDP Sciences, 2016.

[257] Deeptimanta Ojha, Rajesh Kumar Sahoo, and Satyabrata Das. Automatic generation of timetable using firefly algorithm. *International Journal*, 6(4), 2016.

[258] Tsai Duan Lin, Chiun Chieh Hsu, and Li Fu Hsu. Optimization by ant colony hybrid local search for online class constrained bin packing problem. In *Applied Mechanics and Materials*, volume 311, pages 123–128. Trans Tech Publ, 2013.

[259] A Selvakumar and G Gunasekaran. A novel approach of load balancing and task scheduling using ant colony optimization algorithm. *International Journal of Software Innovation (IJSI)*, 7(2):9–20, 2019.

[260] Xiaokun Duan, Bo Wu, Youmin Hu, Jie Liu, and Jing Xiong. An improved artificial bee colony algorithm with maxtf heuristic rule for two-sided assembly

line balancing problem. *Frontiers of Mechanical Engineering*, 14(2):241–253, 2019.

[261] Erfan Babaee Tirkolaee, Mehdi Alinaghian, Ali Asghar Rahmani Hosseinabadi, Mani Bakhshi Sasi, and Arun Kumar Sangaiah. An improved ant colony optimization for the multi-trip capacitated arc routing problem. *Computers & Electrical Engineering*, 77:457–470, 2019.

[262] Xinye Chen, Ping Zhang, Guanglong Du, and Fang Li. Ant colony optimization based memetic algorithm to solve bi-objective multiple traveling salesmen problem for multi-robot systems. *IEEE Access*, 6:21745–21757, 2018.

[263] Tatyana Levanova and Alexander Gnusarev. Development of ant colony optimization algorithm for competitive p-median facility location problem with elastic demand. In *International Conference on Mathematical Optimization Theory and Operations Research*, pages 68–78. Springer, 2019.

[264] Zilong Zhuang, Zizhao Huang, Zhiyao Lu, Liangxun Guo, Qi Cao, and Wei Qin. An improved artificial bee colony algorithm for solving open shop scheduling problem with two sequence-dependent setup times. *Procedia CIRP*, 83:563–568, 2019.

[265] Marilyn Bello, Rafael Bello, Ann Nowé, and María M García-Lorenzo. A method for the team selection problem between two decision-makers using the ant colony optimization. In *Soft Computing Applications for Group Decision-making and Consensus Modeling*, pages 391–410. Springer, 2018.

[266] Ann Ahu Aksut. *Population-based ant colony optimization for multivariate microaggregation*. Nova Southeastern University, 2013.

[267] Min Kong, Peng Tian, and Yucheng Kao. A new ant colony optimization algorithm for the multidimensional knapsack problem. *Computers & Operations Research*, 35(8):2672–2683, 2008.

[268] Munirah Mazlan, Mokhairi Makhtar, Ahmad Firdaus Khair Ahmad Khairi, and Mohamad Afendee Mohamed. University course timetabling model using ant colony optimization algorithm approach. *Indonesian Journal of Electrical Engineering and Computer Science*, 13(1):72–76, 2019.

[269] Retno Subekti, ER Sari, and R Kusumawati. Ant colony algorithm for clustering in portfolio optimization. In *Journal of Physics: Conference Series*, volume 983, page 012096. IOP Publishing, 2018.

[270] Lingyan Lv, Chao Gao, Jianjun Chen, Liang Luo, and Zili Zhang. Physarum-based ant colony optimization for graph coloring problem. In *International Conference on Swarm Intelligence*, pages 210–219. Springer, 2019.

[271] T Warren Liao and Poan Su. Parallel machine scheduling in fuzzy environment with hybrid ant colony optimization including a comparison of fuzzy number ranking methods in consideration of spread of fuzziness. *Applied Soft Computing*, 56:65–81, 2017.

[272] Chen-Yang Cheng, Yin-Yann Chen, Tzu-Li Chen, and John Jung-Woon Yoo. Using a hybrid approach based on the particle swarm optimization and ant

colony optimization to solve a joint order batching and picker routing problem. *International Journal of Production Economics*, 170:805–814, 2015.

[273] Ting Zhang, Xintong Yang, Qingxin Chen, Liping Bai, and Wenge Chen. Modified aco for home health care scheduling and routing problem in chinese communities. In *2018 IEEE 15th International Conference on Networking, Sensing and Control (ICNSC)*, pages 1–6. IEEE, 2018.

[274] A Steven, Gatot Fatwanto Hertono, and Bevina Desjwiandra Handari. Clustered stocks weighting with ant colony optimization in portfolio optimization. In *AIP Conference Proceedings*, volume 2023, page 020204. AIP Publishing, 2018.

[275] Arnaud Laurent and Nathalie Klement. Bin packing problem with priorities and incompatibilities using pso: application in a health care community. 2019.

[276] Neha Sethi, Surjit Singh, and Gurvinder Singh. Improved mutation-based particle swarm optimization for load balancing in cloud data centers. In *Harmony Search and Nature Inspired Optimization Algorithms*, pages 939–947. Springer, 2019.

[277] Emel Kızılkaya Aydoğan, Yılmaz Delice, Uğur Özcan, Cevriye Gencer, and Özkan Bali. Balancing stochastic u-lines using particle swarm optimization. *Journal of Intelligent Manufacturing*, 30(1):97–111, 2019.

[278] Yannis Marinakis, Magdalene Marinaki, and Athanasios Migdalas. A multi-adaptive particle swarm optimization for the vehicle routing problem with time windows. *Information Sciences*, 481:311–329, 2019.

[279] Vahid Mahmoodian, Armin Jabbarzadeh, Hassan Rezazadeh, and Farnaz Barzinpour. A novel intelligent particle swarm optimization algorithm for solving cell formation problem. *Neural Computing and Applications*, 31(2):801–815, 2019.

[280] Shanchen Pang, Tan Li, Feng Dai, and Meng Yu. Particle swarm optimization algorithm for multi-salesman problem with time and capacity constraints. *Applied Mathematics & Information Sciences*, 7(6):2439, 2013.

[281] IA Osinuga, AA Bolarinwa, and LA Kazakovtsev. A modified particle swarm optimization algorithm for location problem. In *IOP Conference Series: Materials Science and Engineering*, volume 537, page 042060. IOP Publishing, 2019.

[282] Maroua Nouiri, Abdelghani Bekrar, Abderezak Jemai, Smail Niar, and Ahmed Chiheb Ammari. An effective and distributed particle swarm optimization algorithm for flexible job-shop scheduling problem. *Journal of Intelligent Manufacturing*, 29(3):603–615, 2018.

[283] Walaa H El-Ashmawi, Ahmed F Ali, and Mohamed A Tawhid. An improved particle swarm optimization with a new swap operator for team formation problem. *Journal of Industrial Engineering International*, 15(1):53–71, 2019.

[284] Orlando Durán, Luis Pérez, and Antonio Batocchio. Optimization of modular structures using particle swarm optimization. *Expert Systems with Applications*, 39(3):3507–3515, 2012.

[285] Xuan Ma and Yufeng Zhang. A particle swarm optimization based on many-objective for multiple knapsack problem. In *2019 14th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 260–265. IEEE, 2019.

[286] Thatchai Thepphakorn and Pupong Pongcharoen. Variants and parameters investigations of particle swarm optimisation for solving course timetabling problems. In *International Conference on Swarm Intelligence*, pages 177–187. Springer, 2019.

[287] Laith Mohammad Abualigah, Ahamad Tajudin Khader, and Essam Said Hanandeh. A new feature selection method to improve the document clustering using particle swarm optimization algorithm. *Journal of Computational Science*, 25:456–466, 2018.

[288] RJ Kuo, SY Hong, and YC Huang. Integration of particle swarm optimization-based fuzzy neural network and artificial neural network for supplier selection. *Applied Mathematical Modelling*, 34(12):3976–3990, 2010.

[289] Ze-shu RAO, Wan-ying ZHU, and Kai ZHANG. Solving graph coloring problem using parallel discrete particle swarm optimization on cuda. *DEStech Transactions on Engineering and Technology Research*, (amsm), 2017.

[290] SH Pakzad-Moghaddam. A lévy flight embedded particle swarm optimization for multi-objective parallel-machine scheduling with learning and adapting considerations. *Computers & Industrial Engineering*, 91:109–128, 2016.

[291] Chananes Akjiratikarl, Pisal Yenradee, and Paul R Drake. Pso-based algorithm for home care worker scheduling in the uk. *Computers & Industrial Engineering*, 53(4):559–583, 2007.

[292] Efim Bronshtein and Olga Kondrateva. The decision support of the securities portfolio composition based on the particle swarm optimization. In *7th Scientific Conference on Information Technologies for Intelligent Decision Making Support (ITIDS 2019)*. Atlantis Press, 2019.

[293] Mohamed Abdel-Basset, Gunasekaran Manogaran, Laila Abdel-Fatah, and Seyedali Mirjalili. An improved nature inspired meta-heuristic algorithm for 1-d bin packing problems. *Personal and Ubiquitous Computing*, 22(5-6):1117–1132, 2018.

[294] Deepak Garg and Pardeep Kumar. Evaluation and improvement of load balancing using proposed cuckoo search in cloudsim. In *International Conference on Advanced Informatics for Computing Research*, pages 343–358. Springer, 2019.

[295] Zixiang Li, Nilanjan Dey, Amira S Ashour, and Qiuhua Tang. Discrete cuckoo search algorithms for two-sided robotic assembly line balancing problem. *Neural Computing and Applications*, 30(9):2685–2696, 2018.

[296] Jon Henly Santillan, Samantha Tapucar, Cinmayii Manliguez, and Vicente Calag. Cuckoo search via lévy flights for the capacitated vehicle routing problem. *Journal of Industrial Engineering International*, 14(2):293–304, 2018.

[297] Bouchra Karoum and Youssef Bouazza Elbenani. Discrete cuckoo search algorithm for solving the cell formation problem. *International Journal of Manufacturing Research*, 14(3):245–264, 2019.

[298] Asri Bekti Pratiwi, Nur Faiza, and Edi Edi Winarko. Penerapan cuckoo search algorithm (csa) untuk menyelesaikan uncapacitated facility location problem (uflp). *Contemporary Mathematics and Applications*, 1(1):34–45, 2019.

[299] Rakesh Kumar Phanden, Zuzana Palková, and Rahul Sindhwani. A framework for flexible job shop scheduling problem using simulation-based cuckoo search. *Advances in Industrial and Production Engineering: Select Proceedings of FLAME 2018*, page 247, 2019.

[300] Hayam G Wahdan, Sally S Kassem, and Hisham ME Abdelsalam. Product modularization using cuckoo search algorithm. In *International Conference on Operations Research and Enterprise Systems*, pages 20–34. Springer, 2016.

[301] Ezreen Farina Shair, SY Khor, AR Abdullah, HI Jaafar, NZ Saharuddin, and AF Zainal Abidin. Cuckoo search approach for cutting stock problem. *International Journal of Information and Electronics Engineering*, 5(2):138, 2015.

[302] Krishna Gopal Dhal, Arunita Das, Swarnajit Ray, and Sanjoy Das. A clustering based classification approach based on modified cuckoo search algorithm. *Pattern Recognition and Image Analysis*, 29(3):344–359, 2019.

[303] G Kanagaraj, SG Ponnambalam, and N Jawahar. Reliability-based total cost of ownership approach for supplier selection using cuckoo-inspired hybrid algorithm. *The International Journal of Advanced Manufacturing Technology*, 84(5-8):801–816, 2016.

[304] Claus Aranha, Keita Toda, and Hitoshi Kanoh. Solving the graph coloring problem using cuckoo search. In *International Conference on Swarm Intelligence*, pages 552–560. Springer, 2017.

[305] Elham Shadkam, Reza Delavari, Farzad Memariani, and Morteza Poursaleh. Portfolio selection by the means of cuckoo optimization algorithm. *International Journal on Computational Science & Application*, 2015.

[306] Luocheng Shen, Jiazhou Li, Yan Wu, Zhenyu Tang, and Yi Wang. Optimization of artificial bee colony algorithm based load balancing in smart grid cloud. In *2019 IEEE Innovative Smart Grid Technologies-Asia (ISGT Asia)*, pages 1131–1134. IEEE, 2019.

[307] Jing Xiong, Xiaokun Duan, and Erhua Wang. A hybrid artificial bee colony algorithm for balancing two-sided assembly line with assignment constraints. In *Journal of Physics: Conference Series*, volume 1303, page 012145. IOP Publishing, 2019.

[308] M Davoodi, M Malekpour Golsefidi, and MS Mesgari. A hybrid optimization method for vehicle routing problem using artificial bee colony and genetic algorithm. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42:293–297, 2019.

[309] Adinarayanan Arunagiri, Uthayakumar Marimuthu, Prabhakaran Gopalakrishnan, Adam Slota, Jerzy Zajac, and Maheandera Prabu Paulraj. Sustainability formation of machine cells in group technology systems using modified artificial bee colony algorithm. *Sustainability*, 10(1):42, 2018.

[310] Shin Siang Choong, Li-Pei Wong, and Chee Peng Lim. An artificial bee colony algorithm with a modified choice function for the traveling salesman problem. *Swarm and evolutionary computation*, 44:622–635, 2019.

[311] José A Delgado-Osuna, Manuel Lozano, and Carlos García-Martínez. An alternative artificial bee colony algorithm with destructive–constructive neighbourhood operator for the problem of composing medical crews. *Information Sciences*, 326:215–226, 2016.

[312] Shima Sabet, Mohammad Shokouhifar, and Fardad Farokhi. A discrete artificial bee colony for multiple knapsack problem. *International Journal of Reasoning-based Intelligent Systems*, 5(2):88–95, 2013.

[313] Changsheng Zhang, Dantong Ouyang, and Jiaxu Ning. An artificial bee colony approach for clustering. *Expert Systems with Applications*, 37(7):4761–4767, 2010.

[314] Francisco J Rodriguez, Manuel Lozano, Carlos García-Martínez, and Jonathan D GonzáLez-Barrera. An artificial bee colony algorithm for the maximally diverse grouping problem. *Information Sciences*, 230:183–196, 2013.

[315] Baris Yuce and Ernesto Mastrocinque. A hybrid approach using the bees algorithm and fuzzy-ahp for supplier selection. In *Handbook of research on advanced computational techniques for simulation-based engineering*, pages 171–194. IGI Global, 2016.

[316] Kui Chen and Hitoshi Kanoh. A discrete artificial bee colony algorithm based on similarity for graph coloring problems. In *International Conference on Theory and Practice of Natural Computing*, pages 73–84. Springer, 2016.

[317] Erdal Caniyilmaz, Betül Benli, and Mehmet S Ilkay. An artificial bee colony algorithm approach for unrelated parallel machine scheduling with processing set restrictions, job sequence-dependent setup times, and due date. *The International Journal of Advanced Manufacturing Technology*, 77(9-12):2105–2115, 2015.

[318] Zhonghua Li and Zijing Zhou. An effective batching method based on the artificial bee colony algorithm for order picking. In *2013 Ninth International Conference on Natural Computation (ICNC)*, pages 386–391. IEEE, 2013.

[319] TD Maydina, GF Hertono, and BD Handari. Implementation of agglomerative clustering and modified artificial bee colony algorithm on stock portfolio optimization with possibilistic constraints. In *AIP Conference Proceedings*, volume 2168, page 020030. AIP Publishing, 2019.

[320] Chuanxin Zhao, Lin Jiang, and Kok Lay Teo. A hybrid chaos firefly algorithm for three-dimensional irregular packing problem. *Journal of Industrial & Management Optimization*, pages 147–157, 2018.

[321] Gundipika Kaur and Kiranbir Kaur. An adaptive firefly algorithm for load balancing in cloud computing. In *Proceedings of Sixth International conference on Soft Computing for Problem Solving*, pages 63–72. Springer, 2017.

[322] Lixia Zhu, Zeqiang Zhang, and Yi Wang. A pareto firefly algorithm for multi-objective disassembly line balancing problems with hazard evaluation. *International Journal of Production Research*, 56(24):7354–7374, 2018.

[323] Asma M Altabeeb, Abdulqader M Mohsen, and Abdullatif Ghallab. An improved hybrid firefly algorithm for capacitated vehicle routing problem. *Applied Soft Computing*, 84:105728, 2019.

[324] Supriya Ingole and Dinesh Singh. Unequal-area, fixed-shape facility layout problems using the firefly algorithm. *Engineering Optimization*, 49(7):1097–1115, 2017.

[325] Mostafa Mohammadi, Golman Rahmanifar, and GARNA GHASEM KAVEH. Optimization multiple traveling salesman problem by considering the learning effect function in skill and workload balancing of salesman with using the firefly algorithm. 2016.

[326] A Rahmani and SA MirHassani. A hybrid firefly-genetic algorithm for the capacitated facility location problem. *Information Sciences*, 283:70–78, 2014.

[327] Beibei Fan, Wenwei Yang, and Zaifang Zhang. Solving the two-stage hybrid flow shop scheduling problem based on mutant firefly algorithm. *Journal of Ambient Intelligence and Humanized Computing*, 10(3):979–990, 2019.

[328] Adil Baykasoğlu and Fehmi Burcin Ozsoydan. An improved firefly algorithm for solving dynamic multidimensional knapsack problems. *Expert Systems with Applications*, 41(8):3712–3725, 2014.

[329] Abhilash Namdev and BK Tripathy. Scalable rough c-means clustering using firefly algorithm. *International Journal of Computer Science and Business Informatics*, 16(2):1–14, 2016.

[330] Kui Chen and Hitoshi Kanoh. A discrete firefly algorithm based on similarity for graph coloring problems. In *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 65–70. IEEE, 2017.

[331] Latifa Dekhici, Rabeh Redjem, Khaled Belkadi, and Abderrahman El Mhamedi. Discretization of the firefly algorithm for home care. *Canadian Journal of Electrical and Computer Engineering*, 42(1):20–26, 2019.

[332] Hassan Heidari and Laya Neshatizadeh. Stock portfolio-optimization model by mean-semi-variance approach using of firefly algorithm and imperialist competitive algorithm. *International Journal of Business and Development Studies*, 10(1):115–143, 2018.

[333] Ali Husseinzadeh Kashan, Mina Husseinzadeh Kashan, and Somayyeh Karimiyan. A particle swarm optimizer for grouping problems. *Information Sciences*, 252:81–95, 2013.

[334] Yanxin Xu. A novel grouping particle swarm optimization approach for 2d irregular cutting stock problem. *International Journal of Control and Automation*, 9(8):369–380, 2016.

[335] Mingyue Feng, Xianqing Yi, Guohui Li, Shaoxun Tang, and He Jun. A grouping particle swarm optimization algorithm for flexible job shop scheduling problem. In *2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, volume 1, pages 332–336. IEEE, 2008.

[336] Weian Guo, Ming Chen, Lei Wang, Yanfen Mao, and Qidi Wu. A survey of biogeography-based optimization. *Neural Computing and Applications*, 28(8):1909–1926, 2017.

[337] Fuqing Zhao, Shuo Qin, Yi Zhang, Weimin Ma, Chuck Zhang, and Houbin Song. A two-stage differential biogeography-based optimization algorithm and its performance analysis. *Expert Systems with Applications*, 115:329–345, 2019.

[338] Michael Mutingi and Charles Mbohwa. Grouping genetic algorithms: Advances for real-world grouping problems. In *Grouping Genetic Algorithms*, pages 45–66. Springer, 2017.

[339] Ethel Mokotoff. Parallel machine scheduling problems: A survey. *Asia-Pacific Journal of Operational Research*, 18(2):193, 2001.

[340] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. Elsevier, 1979.

[341] Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Recent developments in deterministic sequencing and scheduling: A survey. In *Deterministic and stochastic scheduling*, pages 35–73. Springer, 1982.

[342] Luis Fanjul-Peyro and Rubén Ruiz. Iterated greedy local search methods for unrelated parallel machine scheduling. *European Journal of Operational Research*, 207(1):55–69, 2010.

[343] Jan Karel Lenstra, David B Shmoys, and Eva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[344] James Bruno, Edward G Coffman Jr, and Ravi Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17(7):382–387, 1974.

[345] Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM (JACM)*, 23(2):317–327, 1976.

[346] Oscar H Ibarra and Chul E Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM (JACM)*, 24(2):280–289, 1977.

[347] Prabuddha De and Thomas E Morton. Scheduling to minimize makespan on unequal parallel processors. *Decision Sciences*, 11(4):586–602, 1980.

[348] Ernest Davis and Jeffrey M Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM (JACM)*, 28(4):721–736, 1981.

[349] Ci N Potts. Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics*, 10(2):155–164, 1985.

[350] AMA Hariri and Chris N Potts. Heuristics for scheduling unrelated parallel machines. *Computers & operations research*, 18(3):323–331, 1991.

[351] Steef L van de Velde. Duality-based algorithms for scheduling unrelated parallel machines. *ORSA Journal on Computing*, 5(2):192–205, 1993.

[352] CA Glass, CN Potts, and P Shade. Unrelated parallel machine scheduling using local search. *Mathematical and Computer Modelling*, 20(2):41–52, 1994.

[353] Nanda Piersma and Wim van Dijk. A local search heuristic for unrelated parallel machine scheduling with efficient neighborhood search. *Mathematical and Computer Modelling*, 24(9):11–19, 1996.

[354] Silvano Martello, François Soumis, and Paolo Toth. Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete applied mathematics*, 75(2):169–188, 1997.

[355] Bharatendu Srivastava. An effective heuristic for minimising makespan on unrelated parallel machines. *Journal of the Operational Research Society*, 49(8):886–894, 1998.

[356] Ethel Mokotoff and Philippe Chrétienne. A cutting plane algorithm for the unrelated parallel machine scheduling problem. *European Journal of Operational Research*, 141(3):515–525, 2002.

[357] Maria Serna and Fatos Xhafa. Approximating scheduling unrelated parallel machines in parallel. *Computational Optimization and Applications*, 21(3):325–338, 2002.

[358] Ethel Mokotoff and JL Jimeno. Heuristics based on partial enumeration for the unrelated parallel processor scheduling problem. *Annals of Operations Research*, 117(1-4):133–150, 2002.

[359] Yunsong Guo, Andrew Lim, Brian Rodrigues, and Liang Yang. Minimizing the makespan for unrelated parallel machines. *International Journal on Artificial Intelligence Tools*, 16(03):399–415, 2007.

[360] Michele Pfund, John W Fowler, and Jatinder ND Gupta. A survey of algorithms for single and multi-objective unrelated parallel-machine deterministic scheduling problems. *Journal of the Chinese Institute of Industrial Engineers*, 21(3):230–241, 2004.

[361] VS Kumar, Madhav V Marathe, Srinivasan Parthasarathy, and Aravind Srinivasan. A unified approach to scheduling on unrelated parallel machines. *Journal of the ACM (JACM)*, 56(5):28, 2009.

[362] Martin Gairing, Burkhard Monien, and Andreas Woclaw. A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. *Theoretical Computer Science*, 380(1-2):87–99, 2007.

[363] Aleksei V Fishkin, Klaus Jansen, and Monaldo Mastrolilli. Grouping techniques for scheduling problems: Simpler and faster. *Algorithmica*, 51(2):183–199, 2008.

[364] YK Lin, ME Pfund, and JW Fowler. Minimizing makespans for unrelated parallel machine scheduling problems. In *Service Operations, Logistics and Informatics, 2009. SOLI'09. IEEE/INFORMS International Conference on*, pages 107–110. IEEE, 2009.

[365] P Sivasankaran, T Sornakumar, and R Panneerselvam. Efficient heuristic to minimize makespan in single machine scheduling problem with unrelated parallel machines. *Intelligent Information Management*, 2(3):188–198, 2010.

[366] Panneerselvam Sivasankaran, Thambu Sornakumar, and Ramasamy Panneerselvam. Design and comparison of simulated annealing algorithm

and grasp to minimize makespan in single machine scheduling with unrelated parallel machines. 2010.

[367] Christoforos Charalambous, Krzysztof Fleszar, and Khalil S Hindi. A hybrid searching method for the unrelated parallel machine scheduling problem. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 230–237. Springer, 2010.

[368] Luis Fanjul-Peyro and Rubén Ruiz. Size-reduction heuristics for the unrelated parallel machines scheduling problem. *Computers & Operations Research*, 38(1):301–309, 2011.

[369] Yang-Kuei Lin, Michele E Pfund, and John W Fowler. Heuristics for minimizing regular performance measures in unrelated parallel machine scheduling problems. *Computers & Operations Research*, 38(6):901–916, 2011.

[370] David B Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical programming*, 62(1-3):461–474, 1993.

[371] Evgeny V Shchepin and Nodari Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters*, 33(2):127–133, 2005.

[372] Klaus Jansen and Lorant Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. *Mathematics of Operations Research*, 26(2):324–338, 2001.

[373] Panneerselvam Sivasankaran, Thambu Sornakumar, and Ramasamy Panneerselvam. Design and comparison of simulated annealing algorithm and grasp to minimize makespan in single machine scheduling with unrelated parallel machines. *Intelligent Information Management*, 2(07):406, 2010.

[374] Marco Ghirardi and Chris N Potts. Makespan minimization for scheduling unrelated parallel machines: A recovering beam search approach. *European Journal of Operational Research*, 165(2):457–467, 2005.

[375] Yang-Kuei Lin, Michele E Pfund, and John W Fowler. Heuristics for minimizing regular performance measures in unrelated parallel machine scheduling problems. *Computers & Operations Research*, 38(6):901–916, 2011.

[376] Octavio Ramos-Figueroa, Marcela Quiroz-Castellanos, Efrén Mezura-Montes, and Rupak Kharel. Variation operators for grouping genetic algorithms: A review. *Swarm and Evolutionary Computation*, 2020.

[377] MHMA Jahromi, R Jafari, and A Shamsi. Solving fms assignment problem with grouping genetic algorithm. *International Journal of Research in Industrial Engineering*, 1(3):60–68, 2012.

[378] Rhydian Lewis and Ben Paechter. Finding feasible timetables using group-based operators. *IEEE Transactions on Evolutionary Computation*, 11(3):397–413, 2007.

[379] Felipe Arenales Santos and Alexandre CB Delbem. Grouping genetic algorithm with efficient data structures for the university course timetabling problem. *PATAT 2010*, page 542.

[380] L'udmila Jánošíková and Patrik Vasilovskỳ. Grouping genetic algorithm for the capacitated p-median problem. In *2017 International Conference on Information and Digital Technologies (IDT)*, pages 152–159. IEEE, 2017.

[381] Itziar Landa-Torres, Javier Del Ser, Sancho Salcedo-Sanz, Sergio Gil-Lopez, José Antonio Portilla-Figueras, and Oscar Alonso-Garrido. A comparative study of two hybrid grouping evolutionary techniques for the capacitated p-median problem. *Computers & Operations Research*, 39(9):2214–2222, 2012.

[382] Kim-Fung Man, Kit Sang Tang, and Sam Kwong. *Genetic algorithms: concepts and designs*. Springer Science & Business Media, 2001.

[383] Thomas Back. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the first IEEE conference on evolutionary computation. IEEE World Congress on Computational Intelligence*, pages 57–62. IEEE, 1994.

[384] Tobias Blickle. Tournament selection. *Evolutionary computation*, 1:181–186, 2000.

[385] Doug Hains. *Structure in combinatorial optimization and its effect on heuristic performance*. PhD thesis, Colorado State University. Libraries, 2013.

[386] Jorge Kanda, Andre de Carvalho, Eduardo Hruschka, Carlos Soares, and Pavel Brazdil. Meta-learning to select the best meta-heuristic for the traveling salesman problem: A comparison of meta-features. *Neurocomputing*, 205:393–406, 2016.

[387] Marco Chiarandini, Luis Paquete, Mike Preuss, and Enda Ridge. Experiments on metaheuristics: Methodological overview and open issues. Technical report, Technical Report DMF-2007-03-003, The Danish Mathematical Society, Denmark, 2007.

[388] Kenneth Sörensen, Marc Sevaux, and Fred Glover. A history of metaheuristics. *Handbook of heuristics*, pages 1–18, 2018.

[389] John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[390] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.

[391] Kate A Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6, 2009.

[392] Pavel Brazdil and Christophe Giraud-Carrier. Metalearning and algorithm selection: progress, state of the art and introduction to the 2018 special issue, 2018.

[393] Erik Pitzer and Michael Affenzeller. A comprehensive survey on fitness landscape analysis. In *Recent advances in intelligent engineering systems*, pages 161–191. Springer, 2012.

[394] Hui Lu, Rongrong Zhou, Zongming Fei, and Chongchong Guan. Spatial-domain fitness landscape analysis for combinatorial optimization. *Information Sciences*, 472:126–144, 2019.

[395] Khulood Alyahya and Jonathan E Rowe. Landscape analysis of a class of np-hard binary packing problems. *Evolutionary computation*, 27(1):47–73, 2019.

[396] Zhihui Wang, Bryan O'Gorman, Tony T Tran, Eleanor G Rieffel, Jeremy Frank, and Minh Do. An investigation of phase transitions in single-machine scheduling problems. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*, 2017.

[397] Bastien Chopard and Marco Tomassini. Phase transitions in combinatorial optimization problems. In *An Introduction to Metaheuristics for Optimization*, pages 171–189. Springer, 2018.

[398] Toby Walsh and John Slaney. Backbones in optimization and approximation. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.

[399] Kate Smith-Miles and Leo Lopes. Measuring instance difficulty for combinatorial optimization problems. *Computers & Operations Research*, 39(5):875–889, 2012.

[400] Marcela Quiroz-Castellanos. *Caracterización del proceso de optimización de algoritmos heurísticos aplicados al problema de empacado de objetos en contenedores*. PhD thesis, Instituto Tecnológico de Tijuana, 2014.

[401] Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[402] Tugrul Bayraktar, Mehmet Emin Aydin, and Muharrem Dugenci. A memory-integrated artificial bee algorithm for 1-d bin packing problems. In *Proc. CIE IMSS*, pages 1023–1034, 2014.