symmetry

MDPI

Systematic Review

Systematic Literature Review of Optimization Algorithms for $P||C_{max}$ Problem

Dragutin Ostojić [1], Dušan Ramljak [2,*], Andrija Urošević [3], Marija Jolović [1], Radovan Drašković [1], Jainil Kakka [2], Tatjana Jakšić Krüger [4] and Tatjana Davidović [4]

[1] Faculty of Science, Department of Mathematics and Informatics, University of Kragujevac, Kragujevac, 34000, Serbia; dragutin.ostojic@pmf.kg.ac.rs (D.O.); marija.jolovic@pmf.kg.ac.rs (M.J.); radovan.draskovic@pmf.kg.ac.rs (R.D.)
[2] School of Professional Graduate Studies at Great Valley, The Pennsylvania State University, Malvern, PA, 19355, USA; dusan@psu.edu (D.R.); jrk6199@psu.edu (J.K.)
[3] Faculty of Mathematics, University of Belgrade, Belgrade, 11000, Serbia; andrija.urosevic@matf.bg.ac.rs
[4] Mathematical Institute, Serbian Academy of Sciences and Arts, Belgrade, 11000, Serbia; tatjana@mi.sanu.ac.rs (T.J.K.); tanjad@mi.sanu.ac.rs (T.D.)
* Correspondence: dusan@psu.edu; Tel.: +1-610-648-3299

Abstract: In the era of open data and open science, it is important that, before announcing their new results, authors consider all previous studies and ensure that they have competitive material worth publishing. To save time, it is popular to replace the exhaustive search of online databases with the utilization of generative Artificial Intelligence (AI). However, especially for problems in niche domains, generative AI results may not be precise enough and sometimes can even be misleading. A typical example is $P||C_{max}$, an important scheduling problem studied mainly in a wider context of parallel machine scheduling. As there is an uncovered symmetry between $P||C_{max}$ and other similar optimization problems, it is not easy for generative AI tools to include all relevant results into search. Therefore, to provide the necessary background data to support researchers and generative AI learning, we critically discuss comparisons between algorithms for $P||C_{max}$ that have been presented in the literature. Thus, we summarize and categorize the "state-of-the-art" methods, benchmark test instances, and compare methodologies, all over a long time period. We aim to establish a framework for fair performance evaluation of algorithms for $P||C_{max}$, and according to the presented systematic literature review, we uncovered that it does not exist. We believe that this framework could be of wider importance, as the identified principles apply to a plethora of combinatorial optimization problems.

Keywords: combinatorial optimization algorithms; experimental evaluation; scheduling independent jobs on parallel machines; problem instances; systematic literature review

Academic Editor: Jie Yang

Received: 17 December 2024
Revised: 17 January 2025
Accepted: 20 January 2025
Published:

Citation: Ostojić, D.; Ramljak, D.; Urošević, A.; Jolović, M.; Drašković, R.; Kakka, J.; Jakšić Krüger, T.; Davidović, T. Systematic Literature Review of Optimization Algorithms for $P||C_{max}$ Problem. *Symmetry* **2025**, *1*, 0. https://doi.org/

1. Introduction

Nowadays, artificial Intelligence (AI) and Machine Learning (ML) are an integral part of the intelligent core of almost any business. Disruptive technologies, such as generative AI, Blockchain (BC), the metaverse, etc., are becoming more and more pervasive. Thus, there is a growing need for the involvement of supercomputers for processing huge datasets in real time. Consequently, it is equally important to optimize the utilization of high-performance computing resources, cloud computing, and massively parallel multiprocessor systems. Even on a smaller scale, in mobile computing, there is still a need for efficiently solving scheduling problems. A recently developed systematic difficulty estimation framework [1] could provide an opportunity for a fair comparison of algorithms.

To detect the most efficient optimization algorithm and its required execution time, the framework needs a sufficiently large database containing algorithms and the results of their usage. Numerous optimization algorithms exist for scheduling problems, but there are not nearly enough problem instances on which the algorithms were tested. Accurate and exhaustive comparisons among algorithms in the literature could help enrich the database of instances illustrating algorithms' performance. This database could also be useful for researchers who develop new optimization algorithms to adequately compare algorithms and establish the value of their findings and results.

Nowadays, researchers commonly use generative AI instead of exhaustive searching of online databases. Unfortunately, generative AI results may be imprecise or even misleading, especially for problems in niche domains. As a case study to describe the challenges that arise when comparing different optimization algorithms, we focus on the problem of scheduling a set of independent tasks (jobs) on a set of identical parallel machines, with the goal of minimizing the time required by the last machine to complete its assigned jobs. This time is referred to as *makespan*, and the problem is denoted by $P||C_{max}$ [2,3]. $P||C_{max}$ is an important scheduling problem studied mainly in a wider context of parallel machine scheduling. Solving the $P||C_{max}$ problem efficiently is crucial in various real-world scenarios, such as manufacturing processes, data processing, and distributed computing. Researchers and practitioners use various algorithms and optimization techniques to address the complexities associated with this scheduling problem.

In the 1970s, several optimization algorithms were developed and several equivalent notations were introduced [2,4]. Although the proper notations exist, there has been inconsistent use of acronyms, like Identical Parallel Machines Scheduling (IPMS) or just shortly PMS. The same holds for naming conventions. For example, identical parallel machines scheduling problems with the objective function of minimizing makespan (C_{max}) [5], the problem of static scheduling of independent tasks on homogeneous multiprocessors [6], and static homogeneous multi processor scheduling problem (MPSP) [7].

The naming conventions also differ depending on the community of researchers using the problem. In the Operations Research community literature, the $P||C_{max}$ problem is often named IPMS [8]. In the AI community, it is often called the Multi-Way Number Partitioning (MWNP) problem [9]. However, MWNP has a different formulation and is an isomorphic problem to $P||C_{max}$. The second isomorphic problem to $P||C_{max}$ is Min-Max (one-dimensional) Bin Packing Problem (MMBPP). In this variant of Bin Packing Problem (BPP), the number of bins is fixed and the goal is to find a minimum bin capacity that accommodates all items. The problem can also be found under different names, from only a special variant of BPP [10] to BPP-2 [11]. However, this is exactly the same problem as $P||C_{max}$, actually the variant with a fixed number of machines ($P_m||C_{max}$), because there is no practical difference between the variants for reasonably big values of m.

By identifying the equivalence between similar optimization problems, $P||C_{max}$, MWNP, MMBPP, multiple Knapsack problem, multiple subset sum, etc. [12], we have uncovered a valuable symmetry that has been leveraged in $P||C_{max}$ optimization algorithms. Additionally, in combinatorial optimization problems, symmetries can lead to redundant computations as symmetric solutions are essentially equivalent. The search involving symmetric solutions should be avoided by implementing symmetry-breaking constraints in such a way that symmetric solutions are efficiently eliminated from the search space. It is relatively easy to overcome the difficulties related to naming or symmetry and to identify research papers that focus on the $P||C_{max}$ problem through the exhaustive time-consuming search of online databases. It is much more difficult is to obtain the non-confounding results using generative AI, to establish the taxonomy of optimization

algorithms, and to standardize problem instances that were generated. Our goal is to contribute toward resolving this difficulty.

In the preparation for this paper, we decided to search for exact, heuristic, and metaheuristic algorithms. As it has been proven that the $P||C_{max}$ problem is strongly $\mathcal{NP}$-hard [2], it can be assumed that there always exists a problem instance that is hard to solve regardless of the algorithm choice. At the same time, a special case related to this problem is where the machine number that is equal to 2 is "the easiest hard problem" [13]. Therefore, comparisons need to be experimental and to include a wide range of problem instances.

Different optimization algorithms do not have the same performance on all problem instances. One group of instances might be easy for one optimization algorithm, but difficult for the other and vice versa [14]. It could be said that one optimization algorithm is better if it is able to efficiently solve a wider range of problem instances. In a recent study [15], the issues arising in choosing a group of instances for algorithm comparisons and how to generate problem instances were discussed. To provide a fair comparison of algorithms it is very important that experimental settings are identical or that analogies could be established and all the assumptions could be replicated. Thus, it is very important to publicly share the problem instances, or give precise instructions on how to generate them including to provide the values of parameters, like random seed.

Currently, no universal standard exists on how to compare the algorithms working on $P||C_{max}$ problem. Various problem instances were proposed over time, but there is no universal standard set of benchmark instances that everyone could use. We performed a Systematic Literature Review of Optimization Algorithms for the $P||C_{max}$ problem to illustrate that an in-depth analysis of the performance comparison is actually missing. A good algorithm comparison framework is necessary to uncover their strengths and weaknesses. While we have encountered research papers across various avenues that address the literature surrounding this specified problem, there seems to be a lack of comprehensive synthesis that brings together all the mentioned components (methods, problem instances, comparisons) and builds upon the entirety of existing knowledge. This presents an opportunity to foster a more cohesive understanding of this domain that would allow researchers to assess their contributions easily and in a consistent manner.

The contributions of this paper are as follows.

- Report the results of a systematic literature review (SLR) conducted to identify, extract, evaluate, and synthesize the studies on the $P||C_{max}$ optimization algorithms. Summarize and categorize existing methods;
- Standardize the problem instances that the majority of algorithms were tested on;
- Uncover a comparison methodology for a fair algorithm performance evaluation.

Having in mind the cited guidelines about the organization of systematic literature review papers and the above mentioned goals and contributions, the paper roadmap is based on the following. The main goal is to provide a good framework for the evaluation/comparison of various types of optimization algorithms. It is necessary to present the characteristics of all the algorithms and available instances before analyzing the described evaluation mechanisms and providing the recommendations for their upgrading.

Therefore, the remainder of this paper is structured as follows: Section 2 provides a background on the $P||C_{max}$ problem, solutions, and conducted SLR. Section 3 presents SLR methodology, research questions (RQs), and a high-level overview of the selected studies. The taxonomy of solution methods for the $P||C_{max}$ problem and a summary of findings on the extracted studies are presented in Section 4. It contains the classification of methods, description of main characteristics for each class, brief explanation of each considered algorithm, and the existing similarities between them. In Section 5, the use of problem instances over time is discussed. We review the most commonly used sets, identify

their strengths and weaknesses, and perform their classification. In addition, we propose standardization criteria, select a representative set of standardized instances, and generate an open access repository to store these and any newly proposed benchmark instances. Algorithm comparisons are explored in Section 6. We explain the general methodology for comparing optimization algorithms and how it applies to the comparison of algorithms for the $P||C_{max}$ problem. We review the comparisons presented in the literature and illustrate the resulting ranking of algorithms. Section 7 provides an in-depth analysis, based on the results and study limitations, which draws a roadmap for future research. Our main findings and avenues for future work are summarized in Section 8.

2. $P||C_{max}$ Background

Herein we focus on the $P||C_{max}$ problem that was first introduced in 1959 [16] as "Scheduling Many Processors Which Are Exactly Alike to Finish All Tasks as Soon as Possible". A formal definition of the class of scheduling problems was introduced two decades later in [4] using the three-field notation $\alpha|\beta|\gamma$ to systematically classify scheduling problems. In the three-field notation, α specifies machine environment, β indicates job characteristics, and γ defines optimality criteria. For the $P||C_{max}$, we substitute α with P indicating that machines are identical and work in parallel, β is left empty as jobs are independent, and γ is substituted with C_{max} as the objective is to minimize the maximum completion time (makespan). In [2], a more suitable and mathematically more formal definition for the $P||C_{max}$ problem is introduced. Here, we present combinatorial and mathematical programming formulations of $P||C_{max}$, to introduce a notation that is used systematically and consistently throughout the manuscript.

Definition 1 (**Combinatorial formulation of $P||C_{max}$**). *For a given set $M = \{1, \ldots, m\}$ of $m \in \mathbb{N}$ identical parallel machines, i.e., processors, and a set $J = \{1, \ldots, n\}$ of $n \in \mathbb{N}$ independent jobs, i.e., tasks with positive processing times $p = (p_1, \ldots, p_n)$, the goal is to assign each job to exactly one machine in such a way that the latest machine completion time $C_{max} = \max_{i \in M} C_i$, is minimized, where C_i is the sum of processing times of jobs assigned to machine i.*

Definition 2 (**ILP formulation of $P||C_{max}$ [17]**). *For a given instance (J, p, M), $P||C_{max}$ can be defined in terms of ILP as:*

$$\min \quad C_{max} \tag{1}$$

$$\text{s.t.} \quad \sum_{i \in M} x_{ij} = 1, \qquad j \in J, \tag{2}$$

$$\sum_{j \in J} p_j x_{ij} \leq C_{max}, \qquad i \in M, \tag{3}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in M, j \in J, \tag{4}$$

where binary variable x_{ij} indicates whether job j is assigned to machine i.

A solution of $P||C_{max}$ actually represents the m-partition of the set J. Usually, processing times are natural numbers, but in the literature, these values can sometimes be represented as positive real numbers. The optimal value for C_{max} is usually denoted by C^*_{max}. The proof that $P||C_{max}$ is $\mathcal{NP}$-hard can be found in [2]. It is possible to verify that even $P2||C_{max}$ ($m = 2$) is $\mathcal{NP}$-hard [3,18].

To identify the progress in the development of solution algorithms, as well as to clearly classify the SLR studies, we preliminary divide optimization algorithms that have been applied to the $P||C_{max}$ problem, loosely following [19], into three categories:

1. Exact (E)—provide guarantees of optimality;
2. Heuristic (H)—include constructive, improvement, and approximation algorithms;

3. Metaheuristic (MH)—include general solution frameworks, possibly hybrid algorithms.

More details about the methods in each category and the background of their developments are provided in Section 4.

Table 1 shows what period each SLR study covers, whether it was a $P||C_{max}$ specific survey, which category of optimization algorithms is in the focus of each SLR study, or if any instances (not available—N/A or partially—P) and comparison methods (not available—N/A or partially—P) were covered. "Partially" means that the focus of the SLR was not on presenting which instances and comparison methods exist in the literature, but just mentioning both as a side note. Bolded characteristics signify that SLR covers what we need.

Neither of the SLR studies in the literature specify a period in which the SLR study was covered. Therefore, the table contains the period from the first $P||C_{max}$-related cited reference up to the date the SLR study was published. As expected, not all SLR studies covered all the optimization algorithms. Some of the important methods to solve the $P||C_{max}$ problem are found in [8,20], but majority of methods are scattered throughout other SLR studies. Overwhelmingly, the SLR studies covered a larger group of scheduling problems and devoted a section to discuss the methods that were applied to the $P||C_{max}$ problem. Several SLR studies that focused on the $P||C_{max}$ problem exhaustively covered only a specific group of methods. Currently, there is no SLR study that discusses all methods and all instances that could be found in the literature. Therefore, in Section 5, we explain which groups of instances were widely used and which were unique in the sense that they were used in a few studies and never again. In addition, we collected various sets of test instances, analyze their usability, and made them accessible for the wider research community. The majority of the SLR studies focused on comparing algorithms using Worst Time Complexity (WTC), Worst Space Complexity (WSC), and/or Approximation Ratio (AR). Only a few SLR studies describe algorithm comparison involving results obtained by solving problem instances. To close the gap, Section 6 of our proposed SLR contains not only a detailed description of the existing comparison mechanisms, but also an explanation of a fair performance evaluation framework.

Table 1. Systematic Literature Review Studies.

Paper	Time Period	$P\|\|C_{max}$ Specific	Methods	Instances Used	Comparisons Explained
[21]	1959–1970	no	H	P	P
[22]	1959–1974	no	E, H	N/A	P
[23]	1959–1977	no	H	N/A	N/A
[4]	1959–1979	no	H	N/A	P
[24]	1966–1981	**yes**	H	N/A	P
[18]	1966–1981	no	E, H	N/A	P
[25]	1966–1982	no	H	N/A	P
[26]	1966–1987	no	E, H	P	P
[27]	1969–1987	no	H	N/A	P
[28]	1969–1987	no	E	N/A	P
[29]	1966–1993	no	H, MH	N/A	P
[30]	1959–1994	no	E, H	P	P
[31]	1959–1997	no	E, H	N/A	N/A
[32]	1959–1998	no	E, H	N/A	P
[33]	1959–1999	no	E, H	N/A	P
[34]	1966–2001	no	E, H, MH	N/A	P
[7]	1959–2003	no	MH	P	P
[35]	1959–2004	no	H	N/A	N/A
[36]	1966–2004	no	E, H, MH	N/A	P
[20]	1966–2008	no	E, H, MH	P	P
[37]	1959–2009	no	E, H, MH	N/A	P
[38]	1959–2012	no	E, H, MH	N/A	P
[39]	1966–2013	no	E, H, MH	P	P
[40]	1969–2014	**yes**	E	P	P
[8]	1959–2017	no	E, H, MH	P	P
[41]	1961–2017	**yes**	H	P	P
[42]	1959–2018	**yes**	E, H	P	P
[43]	1982–2022	no	E	N/A	N/A
[44]	1959–2022	no	E, H	P	P
Proposed SLR	1959–2024	**yes**	**All**	**All**	**All**

The proposed SLR focuses on the $P||C_{max}$ problem and all the methods published since the 1950s as there is no SLR that covers them all systematically. Moreover, we include an explanation of all instances and compare methodologies used in the literature. In the next section, we present our SLR methodology.

3. Systematic Literature Review Mapping Methodology

SLRs are the best way to understand the background for developing rigorous research projects. Thus, in many scientific fields there exist standards and guidelines for performing the SLR. The operations research literature provides a taxonomy of SLRs [45] and our SLR fits into the tutorial being selective towards the $P||C_{max}$ problem. Additionally, it is an attempt to understand how to compare optimization algorithms, and how it fits into a broad, comprehensive, computational review. As the optimization community does not have an established SLR standard, we used the methodology based on standards established in closely related fields of software engineering [46]. In this section, we show the SLR methodology by explaining the following:

- Objectives and research questions;
- Search strategy;
- Search criteria;
- Inclusion and exclusion criteria;
- Search and selection procedure;
- Data extraction and synthesis;
- Important characteristics of selected primary studies.

3.1. Objectives and Research Questions

We planned the review process by refining the research objectives into a set of research questions. Our objectives to summarize and categorize the "state-of-the-art" methods, benchmark test instances, and comparison methodology could be accomplished through the following questions:

- RQ1. What are the main characteristics of $P||C_{max}$ optimization methods?
- RQ2. What are the characteristics of problem instances that methods were tested on?
- RQ3. What are the characteristics of comparison methodologies used for performance evaluation of optimization methods identified in RQ1?
- RQ4: Based on RQ3, could a fair algorithm's performance evaluation be defined?

In order to facilitate answering every question, a taxonomy-like characterization of the answers has been identified for the first three questions, and the answer to RQ4 is given in the discussion in Section 7.

3.2. Search Strategy

We utilized the PennState LionSearch tool, Copyright © 2021 The Pennsylvania State University, 201 Old Main, University Park, PA 16802, to explore the available manuscripts. LionSearch is an integrated search engine of books, journal papers, conference papers, and other publications integrated from around 10^3 databases. It includes specialized databases and search engines for operation research, optimization, and applied mathematics disciplines. The tool is provided and maintained by the Pennsylvania State University's Library.

We set up the following parameters for our search strategy: We looked for peer-reviewed articles written in the English language in the following categories: Journal Article, Book Chapter, Conference Proceedings, and Book/eBook. The detailed specification of the search domains included computer science, engineering, mathematics, and general sciences. The search was applied to the full text to ensure that the relevant study's keywords were not missed in the title or abstract. The time period was not specified to ensure that all publications receive the chance to be selected.

3.3. Search Criteria

The search criteria had to include the keywords that were explained in Section 2. Therefore, our query was composed of two strings in the conjunctive normal form. The first should provide general context, while the second string specifies precise terms of interest. An example of a search conducted in the LionSearch is as follows:

((Parallel machine scheduling) OR (Identical Machine Scheduling) OR (Scheduling Problems) OR (independent tasks identical parallel machines) OR (independent jobs identical parallel machines) OR (independent tasks identical parallel processors) OR (independent jobs identical parallel processors)) AND ((P||Cmax) OR (PCmax))

3.4. Inclusion and Exclusion Criteria

The initially retrieved studies from the electronic databases were assessed using the inclusion and exclusion criteria, explained in this subsection.

Inclusion criteria:

- IC1: the language is English;
- IC2: it is relevant to the $P||C_{max}$ problem;
- IC3: it is an empirical research paper, a technical report, a proof of concept, a journal article, a thesis, or a conference paper;
- IC4: it cites or is cited by any of the recognized research studies.

Exclusion criteria:

- EC1: study's focus is not explicitly on scheduling problems related to $P||C_{max}$;
- EC2: the study does not address the optimization;
- EC3: the study does not meet all the inclusion criteria.

All studies that satisfied at least one exclusion criterion were not taken into consideration.

3.5. Search and Selection Procedure

The search was initialized in August 2023 and finalized in December 2024. In the first step, the search returned 3482 results. The majority of these materials contained just a reference to $P||C_{max}$, did not focus on optimization algorithms/instances, or considered a variant that is not relevant. In Step 2, the selection of relevant results was performed and we ended up with a set of 236 documents. The most important component of the selection procedure happened to be the last inclusion criterion. Step 3, was devoted to reading all 236 manuscripts. The full text of each document was reviewed thoroughly by at least two team members. We had weekly meetings to discuss progress and review the findings. During the thorough reading we identified 100 more papers using a snowballing method. That is, we followed which papers cite the relevant papers and identifed papers that are cited by relevant papers. This resulted in 336 papers to assess quality upon.

In the last step, to assess the methodological quality of the primary studies selected for this review, the following quality criteria was adopted:

- QC1: Does the research clearly address any theoretical aspect? (1 or 0);
- QC2: Does the research clearly explain a method? (1 or 0);
- QC3: Are the findings clearly stated? (1 or 0);
- QC4: Based on the findings, is the research valuable? (1 or 0) .

A quality score of 2 was set as the threshold to accept the study for this review. After applying the quality criteria, 261 papers were selected as the primary studies.

3.6. Data Extraction and Synthesis

The data extraction process was conducted by analyzing the selected 261 primary studies. We utilized a predefined extraction form to record the full details of the studies under review and to list the specifics which we could leverage to address motivation, background, and research questions.

3.7. Important Characteristics of the Selected Primary Studies

First, we classified the studies used in this paper according to the type of publication shown in Figure 1.

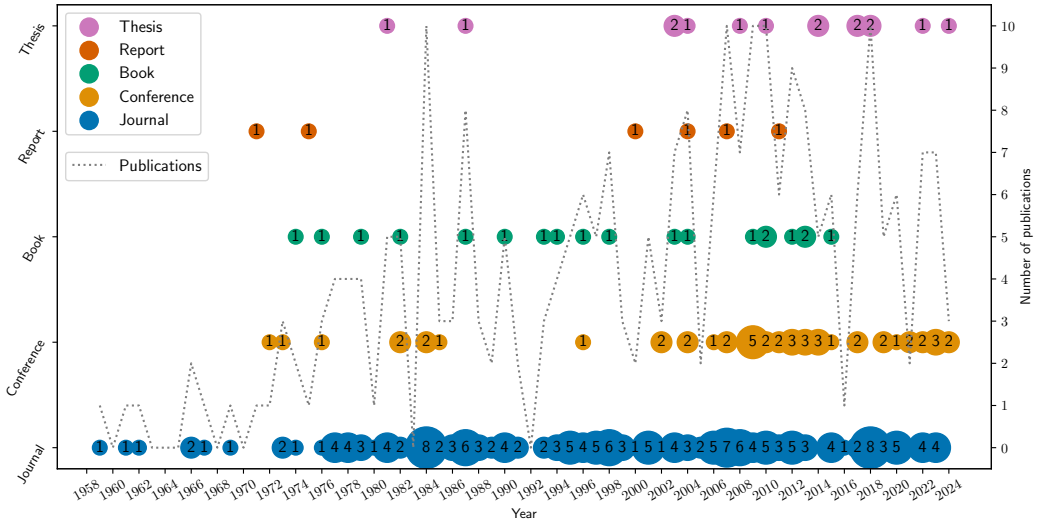

Figure 1. Publication types of selected primary studies.

The figure contains (dotted line) the distribution of studies per year, with their number (maximum 10) indicated on the right side. For each type, the number of studies is written in the circle of the corresponding color.

As shown in Figure 1, the publication years of these references are between 1959 and 2024, with a mean year of 2000.40 and a standard deviation of 15.30. This points to a concentration of references that occur between the year 2000 and the present, with the early 2000s marking a significant surge. The interquartile range falls between 1987 and 2012, indicating a recent focus in the literature.

The majority of the studies, 172 of 261, are journal papers published in the journals from the following fields: discrete and applied mathematics, operations research, and industrial and transportation engineering. Other than clearly identifying the fields, we were not able to extract a useful pattern about the most popular venues for these types of publications. There is a considerate, but significantly smaller number of conference papers, 49 of 261, while books, research reports, and theses appear just sporadically. This distribution reflects a heavy reliance on peer-reviewed journals while also incorporating a variety of other publication types to ensure a comprehensive literature review.

Regarding the impact of publication types, citation counts for these references exhibit considerable variability. The mean number of citations is 455.90, but this is heavily skewed due to a few references receiving high citation numbers, as high as 19,154. The median citation count is much lower at 38, and the interquartile range spans from 10 to 170.50. This indicates that while many references have decent citation counts, there are a few highly influential works significantly affecting the average. The variability underscores the impact of a small number of seminal references. In particular, books were cited 3302.53 times on average (maximum 19155). Journal papers had an average citation count of 288.87 (maximum 7905), and conference papers were cited 33.65 times on average (maximum 441).

In total, 152 studies provide some information related to RQ1, 81 are relevant for RQ2, while for motivation, background, and answering RQ3 and RQ4 all 261 studies were used. We have uncovered that the majority of the recent publications either compare their approaches with the results published more than 25 years ago, or use those algorithms as part of their solution methods. Contrary to all existing SLRs, we systematically cover all the methods, instances, and comparisons. Additionally, we have made all the timeline(s) clearly visible for each research question and identified group, such that any time period or other characteristics of interest can be distinguished and understood.

4. RQ1: Optimization Algorithms

The basic categorization of optimization methods is given in Section 2 and loosely follows the taxonomy defined in [19]. These basic categories explain three fundamentally different approaches to optimization problems. Exact solvers are able to guarantee the optimality of the solution if enough resources are provided. They generally reduce search space as much as they can to overcome the scarcity of resources. Heuristic methods usually provide "good" solutions within a short running time. In general, heuristics are not able to find optimal solutions or to guarantee the quality of the found solution. Contrary to heuristic approaches that are usually specialized, i.e., they use some a priori knowledge about the considered problem, metaheuristics are general-purpose algorithms. They can be applied to various optimization problems and may be viewed as recipes to guide the development of efficient heuristics for a specific optimization problem. Heuristic and metaheuristic optimization algorithms are focused on the fast construction of feasible solutions or on the iterative improvement in the existing solutions. Figure 2 shows the distribution of the selected primary studies that cover each identified optimization algorithm category.

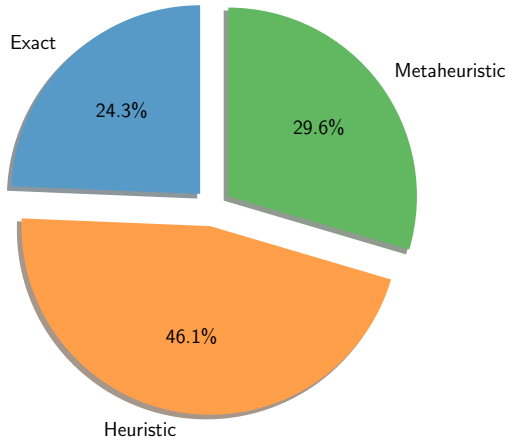

Figure 2. Percentage of different approaches.

As shown in Figure 2, the heuristic methods dominate the other two categories. This pattern might be occurring for two reasons. Heuristics usually provide feasible solutions fast; most providing some insights into the solutions quality and/or complexity of the underlying algorithms. This provides researchers with standards that they can compete on, i.e., publish improvements with respect to them. Metaheuristics have been explored for only 30 years, and experienced a meteoric rise since 2000s; hence, it is unsurprising that they occupy a slightly bigger area than the exact solvers. The latter are still very popular regardless of their limitations. Advantages of exact solvers include but are not limited to their theoretical importance, their applicability to special, easier yet very important, variants of the problem, and the new paradigms that enable the efficient explorations of solution space.

In this section, we explain the important steps for developing $P||C_{max}$ optimization algorithms and divide the categories into groups based on the strategies used to address the $P||C_{max}$ problem. For each group of methods, we present a short history, standardization, and a graph showing the distribution of methods publications over time. When presenting standardization, in each group of optimization algorithms the following table template is used: The columns of the table are labeled: Name of the optimization algorithm (Name); Where and when it is introduced (Reference), known selected characteristics for that optimization algorithm (Known Characteristics); Publications where it is compared with

other algorithms from the same group (Compared with). If any of the information is not provided or not applicable, the corresponding cell contains N/A.

4.1. The Main Issues in Developing Optimization Algorithms

There have been many theoretical discussions about the $P||C_{max}$ problem, its properties, mathematical analysis, relations with other problems, hardness of instances, etc. [2,3,14,47–94]. The majority of the theoretical conclusions are integrated into various optimization algorithms from all groups. In the remainder of this section, we rely on these theoretical results without explicit citations.

For developing any practical optimization algorithm, aiming to reduce its execution time, it is important to conduct the following:

- reduce search space;
- have suitable data structures;
- have efficient rules for the construction/transformation of solutions.

As we want to ensure that our SLR represents a self-contained material, we provide more detail about each of the items mentioned above.

4.1.1. Search Space Reduction

Usually, the first step in reducing the search space is finding good upper and lower bounds. To quickly find a good upper bound, a common practice is to use some constructive heuristic algorithms, which are described in Section 4.3.1. Many lower bound techniques have been developed for $P||C_{max}$ since 1959 [16]. There are no theoretical guarantees that some lower bound technique will be better than others [14,95]; therefore, they are often combined for a better outcome.

To increase the efficiency of the lower bounds calculation, it is usually assumed that jobs are sorted in a non-increasing order of their processing times. The most famous simple lower bound is $L_0 = \lceil \frac{1}{m} \sum_{i=1}^{n} p_i \rceil$ obtained by LP-relaxation rule that allows dividing job processing time among machines [16]. Another simple lower bound denoted by L_1 is calculated as $\max\{L_0, \max_i p_i\}$ [16]. The next simple lower bound is $L_2 = \max\{L_1, p_m + p_{m+1}\}$ [57] obtained by considering the relaxed set of $m+1$ as the biggest jobs, and proved $\frac{L_2}{C^*_{max}} \geq \frac{2}{3}$. By exploiting the fact that there must be a machine processing at least $\nu = \lceil n/m \rceil$ jobs, another simple lower bound is $L_\nu = \sum_{i=n-\nu+1}^{n} p_i$ [57].

A whole class of other lower bounds may be obtained by observing a duality between $P||C_{max}$ and BPP [54]. Assuming that items correspond to jobs, the capacity of each bin corresponds to the BPP-based lower bound L. The number of used bins corresponds to the number of machines m. Finally, L is a valid lower bound for $P||C_{max}$ when the decision procedure of BPP returns yes. The procedures for calculating BPP-based lower bounds L_{HS}, L_3, and L_{FS} are described in [54,57,96].

In addition to simple and BPP-based lower bounds, there are two other lower bounds: L_{GB} [97] and L_θ [57]. To compute general lower bound L_{GB}, an instance of the $P||C_{max}$ problem and machine-ready times (times when each machine can begin processing jobs) are required. Actually, it is possible to compute L_{GB} even in the case when machines are already occupied. On the other hand, L_θ computes upper bound Θ and lower bound θ on the number of jobs per machine.

Each lower bound L_ξ can be lifted using the lifting procedure [95] and lifted lower bound is denoted as $\widetilde{L}_\xi$. There is also the enhanced lifting procedure [14] where the resulting lower bound is denoted as $\overrightarrow{L}_\xi$. The former lifting procedure relaxes instance of $P||C_{max}$ considering $k < m$ machines and the largest $\lambda_k(n) = k\lceil n/m \rceil + \min\{k, n - \lceil n/m \rceil m\}$ jobs. The latter lifting procedure uses a solver for the Subset Sum Problem (SSP) to potentially improve the result of the first lifting procedure.

Some lower-bounding procedures and optimization algorithms perform the search over solution space and require a priori known bounds to reduce that space and, consequently, the execution time. Those particular procedures often use simple, computationally inexpensive lower and upper bounding procedures in order to reduce the solution space [54,57,96]. On the other hand, some procedures depend on the theoretical background of the simpler lower and upper bounds [97]. In addition, quickly found good lower bound can terminate the execution. It can also guarantee optimality if it becomes equal to the objective function value of a previously discovered feasible solution.

Lower-bounding procedures can be very computationally intensive, especially with active lifting procedures. Sometimes it is important to make a balance between a good lower bound and the required execution time for it. Finding a good initial solution can reduce and eventually stop the lower-bounding procedure. On the other hand, a good lower bound can reduce the search space for other solution-searching procedures.

In addition to lower bounds related to makespan, there are also bounds that impose constraints on other solution parameters. These lower bounds can additionally streamline the solution–finding process. That is, given a suboptimal solution S, when searching for an improved solution S', the condition $C^{S'}_{\min} \geq \sum_{i \in m} C^{S'}_i - (C^S_{\max} - 1)(m-1)$ must hold. [42]

There is no full review of lower-bounding procedures and their comparison in the literature. Similar papers include [14,95], and relying on their experiments; hence, it can be concluded that no single procedure dominates for all instances. There is a need for a more detailed comparison of all known lower-bounding procedures on a wider set of instances. However, it is out of the scope of this study. Other than the reduction in search space, lower bounds are important because the gaps with respect to them are used as the performance evaluation in many cited articles.

4.1.2. Data Structures

For a $P||C_{max}$ optimization algorithm efficiency, it is important to define a proper solution representation that ensures quick access to all necessary data. In the studies selected during the SLR process, the following solution representations can be found: *Matrix*, *Permutation*, *List*, and *Path*.

The usual way to represent solutions in the mathematical programming-based approaches is to use binary *assignment matrix* $\mathbf{X}_{m \times n} = (x_{ij})$. For each $i \in M, j \in J$, element x_{ij} of assignment matrix $\mathbf{X}_{m \times n}$ indicates whether job j is assigned to machine i, i.e., $x_{ij} = 1$ when job j is assigned to machine i, and $x_{ij} = 0$ otherwise. The assignment matrix representing a feasible solution for $P||C_{max}$ has exactly one non-zero element in each column, i.e., it belongs to the class of sparse matrices.

Having in mind that each column in the assignment matrix has exactly one non-zero element, storing zeros is not needed. Hence, for each job $j \in J$ one can only store the machine index $\mu(j) \in M$ such that job j is assigned to machine $\mu(j)$, where $\mu : J \mapsto M$ is an assignment function. Solutions can be considered as *permutations of jobs* and these permutations actually define the order of their scheduling. Jobs are taken one by one from the permutation and assigned to the least occupied machine (list scheduling principle). Such a representation is indirect. Although each permutation uniquely defines the corresponding solution, it is necessary to perform additional computations before discovering the makespan value. This representation is more suitable for constructive optimization algorithms than for iterative improvement-based ones.

A solution of $P||C_{max}$ problem could also be represented as an array *Mchn* of m *assignment lists*, one for each machine. For machine $i \in M$, $Mchn[i]$ contains all indices $j \in J$ such that job j is assigned to machine i. The assignment lists can be implemented in various ways: (1) arrays of size n (inactive elements at the end are filled with zeros), (2) singly

and (3) doubly linked lists. Each assignment list can be sorted in a non-increasing order by processing times of jobs. Moreover, assignment lists can be implemented as (4) heaps (facilitate accessing jobs with shortest or longest processing times in constant time) or (5) sets (facilitate accessing/inserting/removing jobs in logarithmic time).

The least intuitive solution representation uses flow networks. The scheduling problem is then formulated as the problem of determining m disjoint *assignment paths* between the source vertex s and the target vertex t such that each job is used exactly once. Each assignment path $p : s \rightsquigarrow t$ represents a machine along which the assigned jobs can be reconstructed.

4.1.3. Construction/Transformation Rules

As already mentioned, optimization algorithms either construct solutions from scratch, or perform transformations on the existing solutions with an aim of improving them. In both cases, some prior knowledge about the problem and the application of learning techniques may increase the efficiency of the underlying procedure. Therefore, special attention in developing $P||C_{max}$ optimization algorithms should be paid to construction and transformation rules. More details are provided in the remainder of this section, where each optimization algorithm is described separately, if applicable.

4.2. Exact Algorithms

The first exact algorithm, presented 1962 [98], is based on dynamic programming. Over the years, many other exact solvers have been proposed. Our SLR revealed that 33 out of 167 selected studies consider the development of exact $P||C_{max}$ solvers. Their main advantage is the optimality guarantee of provided solution, while the disadvantage relates to the large resource requirements. Nonetheless, exact algorithms are still studied from both theoretical and practical points of view.

We have classified the exact $P||C_{max}$ solvers into two groups based on [99,100] and one group of practical significance.

- Exact exponential algorithms (EE);
- Fixed parameter tractable algorithms (FPT);
- Hybrid exact algorithms (HE).

EE algorithms for $\mathcal{NP}$-hard problems are focused on providing as small as possible WTC and WSC. The main characteristic of FPT algorithms is that they have polynomial complexity with respect to the input data, while their exponential dependence on some fixed parameter contributes to WTC. HE algorithms' focus is often on exploring the results of some other type of optimization algorithms to provide optimal solutions with guaranties of optimality in a reasonable amount of time.

The practical importance of HE solvers stems from their adaptation and the ability to explore the results of other types of optimization algorithms as the upper bounds used to reduce the search space and increase the execution speed.

4.2.1. Exponential Exact Algorithms

To the best of our knowledge, all EE solvers for $P||C_{max}$ are presented in Table 2. We include solver names which is based on the technique used and the year of publication.

Table 2. Exponential exact algorithms.

Name	Reference	Known Characteristics	Compared with
DP_{62}	[98] 1962	DP; WTC: $\mathcal{O}(n2^n)$; WSC: $\mathcal{O}(2^n)$	N/A
DP_{66}	[101] 1966	DP; WTC: $\mathcal{O}(\frac{m^n}{n})$	N/A
DP_{76}	[102] 1976	DP; WTC: $\mathcal{O}(min\{2^n, nC\})$	N/A
DP_{82}	[103] 1982	DP; WTC: $\mathcal{O}(nC2^n)$; WSC: $\mathcal{O}(p_{max})$	N/A
DP_{87}	[26] 1987	DP; WTC: $\mathcal{O}(nC^m)$	N/A
DP_{11}	[104] 2011	DP; WTC: $2^{\mathcal{O}(m\sqrt{\lvert I\rvert})}$	N/A
SSM_{13}	[105] 2013	SSM; WTC: $O^*(m^{\frac{m}{2}}(\frac{n}{2})^{m+1})$; WSC: $O^*(m^{\frac{m}{2}}(\frac{n}{2})^{m-2})$	N/A

C—optimal C_{max} value (C^*_{max}). $O^*(\alpha^n) = \mathcal{O}(poly(n)\alpha^n)$. $|I|$—length of the input.

The first developed exact solver [98] belongs to the class of EE solvers. The focus of researchers working on these solvers is improving WTC and WSC, which are therefore much better than those for other exact solvers. Unfortunately, this fact has no practical significance, since these algorithms are not implemented.

Even though many more techniques exist for EE solvers, it can be seen that the first and mainly used technique for $P||C_{max}$ [98] is dynamic programming (DP). Notable EE algorithms later improved DP with techniques like linear waiting costs [101], partition algorithm scheme [102], the principle of inclusion and exclusion [103], recursive formulation [26], and dynamic allocation [104]. In 2013, another EE $P||C_{max}$ solver was presented based on the Sort and Search method (SSM) [105]. The table shows that WTC has been improved consistently and WSC occasionally. Some WTC and WSC values depend on the variables that are indirectly correlated with the size of the input (n and m), and thus, do not have a clear exponential function form. Implementations of EE solvers are rarely found. As shown in Table 2, EE solvers are not directly compared to each other, but there is some cases where they are compared with other optimization algorithms.

Some EE solvers need the assumption of a known C_{max}, i.e., C to establish WTC and WSC in a pseudo-polynomial manner. Thus, the following theoretical result for the version of $P||C_{max}$ with limited machine capacity - $P|C_{max} < C|$ [106] could be relied on. It has been proven the existence of a lower bound on the WTC under the Exponential Time Hypothesis (ETH). Assuming ETH, there is no algorithm that solves this problem with a worst-case time complexity $2^{n^\delta}C^{\mathcal{O}(1)}$ where $\delta \in [0, 1]$, i.e., it cannot be solved in sub-exponential time. Lower bounds for exponential algorithms are studied in [64,107]. From Table 2 it can be seen that EE solvers are approaching this bound.

4.2.2. Fixed Parameter Tractable Algorithms

FPT algorithms focus on solving problems that are computationally hard in general but become tractable when certain parameter k has a small fixed value. In other words, these algorithms aim to provide efficient solutions for specific instances of a problem based on k. Actually, FPT algorithms are exponential only with respect to k, while the size of the input has polynomial influence to the algorithm's running time. More formal definition is that algorithms with running time $f(k)n^c$, for a constant c independent of both n and k, are called FPT algorithms [100]. In such a way, value of k controls the combinatorial explosion.

In 2015, it was proven that the $P||C_{max}$ problem is FPT parameterized by $p_{max} = \max_i p_i$ [62]. Since then, several studies focused on generalizing the results to similar problems [108,109]. Unfortunately, these algorithms cannot be applied in general cases, nor compared to other optimization algorithms. Thus, the practical significance of FPT solvers is very small, and more detailed consideration is out of the scope of this study.

4.2.3. Hybrid Exact Algorithms

The main characteristics of HE algorithms is leveraging the results of other types of optimization algorithms to reduce the search space. Providing optimal solutions, including

guaranties of optimality, in a reasonable amount of time is what contributes to their practical value. HE algorithms identified in our SLR are summarized in Table 3. For the known characteristics field we opted to present what the HE solvers were composed of. As shown in Table 3, HE solvers have been widely used and are still under development. Unlike other exact solvers, they have been compared with other solvers in the same group.

Table 3. Hybrid exact algorithms.

Name	Reference	Known Characteristics	Compared with
N/A	[110,111] 1973	B&B + Lagrange multipliers	N/A
N/A	[112] 1980	KP + Reduction method	N/A
BIN, DM	[57] 1995	BPP, B&B + MS + New LB	N/A
CGA, CKK	[9] 1998	Small m; B&B + LDM	N/A
CP_{04}	[113] 2004	B&B + Cutting plane + Polyhedral theory	N/A
HJ	[14] 2008	B&B + Symmetry-breaking + New LBs	[57]
DIMM	[114] 2008	B&B + Branch and price + Bin. search + SS	[57]
SNP_{ie}, RNP	[115] 2009	Small m; B&B + LDM	[9]
IRNP	[116] 2011	Small m; B&B + LDM + SSP	[115]
MOF	[117] 2013	Small m; B&B + SSP	[116]
BSBCP, CKK_i, RNP_i	[118] 2013	Small m; BPP; B&B + LDM + SSP	[9,116]
BSIBC	[119] 2013	Bin. Search + BPP	[114,118]
SNP_{ess}, H_{14}	[120] 2014	Small m; B&B + LDM + SSP	[117,118]
CIW	[121] 2014	Small m; Iter. weakening + Caching	[117,119,120]
WL, WL′	[8,63] 2017	B&B + Path-related dominance criteria	[14,114]
LCS	[42] 2018	Small m; Iter. weakening + Caching + B&B	[9,115–117,119–121]
KL	[122] 2018	Bin. Search + B&B	[57,114]
AF	[123] 2018	ILP + AF + Reduction criteria + BPP	[8,14]
iAF	[124] 2022	ILP + AF + Graph compress. + BPP + VNS	[123]
DIST	[12] 2023	Decomposition + MKP	[124]

The first HE algorithm for solving $P||C_{max}$ was developed in 1973 [110,111], and is based on Lagrangian multipliers and utilizes Branch and bound (B&B) techniques. The majority of HE algorithms use the B&B techniques with different criteria for branching and bounding searches [8,9,14,42,57,63,110,111,113–118,120,122]. Before branching, HE algorithms tend to tighten the search space as much as possible. For that purpose, lower bounding techniques, various heuristics (e.g., Multi-Subset (MS)), metaheuristics (e.g., Scatter Search (SS) and Variable Neighborhood Search (VNS)) and LP techniques are engaged [57,63,114,122]. Fine balancing between different phases is very important for the best performance.

Some exact solvers for $P||C_{max}$ focus on solving instances with very small m and very big job processing times, which belong to the MWNP formulation. Complete Greedy Algorithm (CGA) [9], besides B&B simply tries all possible greedy ways to construct solution, while Complete Karmarkar–Karp (CKK), presented in the same paper, combines B&B with Largest Differencing Method (LDM) [53] algorithm. Authors used instances with $m \leq 3$, $n \leq 200$ and job processing times up to 10^{10}. In [115], Sequential Number Partitioning with Inclusion Exclusion (SNP_{ie}) and Recursive Number Partitioning (RNP) start with LDM solution and follow the CKK algorithm and B&B strategy, are able to solve instances for m up to five and processing times up to 10^8. IRNP [116] is a version of RNP, improved by using SSP. It can solve instances for m up to 10 and processing times up to 10^{10}. MOF [117] also leverages B&B and SSP with new optimality rule—the principle of weakest-link. It can solve instances for m up to ten and processing times up to 10^{10}. CKK_i and RNP_i [118] are improved versions of CKK and IRNP algorithms. RNP_i are able to solve instances with up to $m = 7$ and with job completion times up to 10^{14}. A new version of the Sequential Number Partitioning with extended Schroeppel and Shamir (SNP_{ess}) was developed in [120]. It generates all first subsets with sums within the lower and upper bounds, and then for each, recursively partitions the remaining numbers $k - 1$ ways. It can solve instances for m up to ten and processing times up to 10^{14}. H_{14} is a hybrid

algorithm that combines CKK, CGA, SNP_{ess}, and MOF. Cached Iterative Weakening (CIW) [121] uses iterative weakening instead of classic B&B, caches feasible subsets, and explores subsets in cardinality order. All algorithms in this paragraph before CIW are anytime algorithms that start with an approximate solution and then improve it until the best partition is found and optimality is proved. In contrast, CIW starts with a lower bound and increases it iteratively until an optimal solution is found, and the first complete partition found is optimal. The LCS algorithm [42] combines the ideas of CIW and the MOF, by search caches only low-cardinality subsets and performs in a two phases. In the first phase, it searches for an upper bound in the way similar to CIW. In the second phase, it uses the B&B technique to either prove the upper bound optimal or find a better optimal solution.

Other than B&B, a few solvers leverage binary search and duality with BPP. In [57], the modification of MF CH (see Section 4.3.1) is used for the BIN algorithm. It is realized by replacing the FFD procedure with exact MTP algorithm [125] with some small modifications and limited number of backtracks to $5 \cdot 10^3$. For starting bounds they used L_2 and result of LPT CH. In the same paper, authors presented a very similar approach called MF, as part of the initial phrase for their B&B DM algorithm. BSBCP works in a similar way, but it uses the BCP algorithm [126] for solving BPP. The DIMM algorithm also uses a similar principle with more advanced initial bounds, and BSIBC [119] uses BPP solver improved by the authors.

Recently, ILP and graph compression techniques were combined with Arc Flow (AF) formulation [123,124]. The resulting algorithms (AF and iAF) show remarkable performance on some $P||C_{max}$ instances, however, they do not provide any feasible solution if they are not able to find the optimal one. Another notable technique [112] reported in 1980 was based on the analogy between $P||C_{max}$ and the Knapsack Problem (KP). Recently, the DIST algorithm [12], based on the decomposition and Multiple Knapsack Problem (MKP), can guarantee the optimality of the provided solution when adequate time limits for subproblems are defined. In all other cases it acts as iterative heuristic and can provide good suboptimal solutions very fast.

4.2.4. Summary of Exact Approaches

To conclude the section on exact approaches, in Figure 3, we present the time distribution of studies in which exact solvers were either developed or compared with other optimization algorithms. From the figure, it is easy to distinguish the development of the exact approaches in any decade of interest. EE solvers were the first developed; however, the most recent study was about a decade ago. FTP solvers were only recently developed with limited practical usability and are not represented in this figure as their focus might only be related to some special variants of the $P||C_{max}$ problem. Finally, HE solvers have been developed since the introduction of the formal $P||C_{max}$ problem definitions in the early 70s. Their development and usage has been intensified recently.

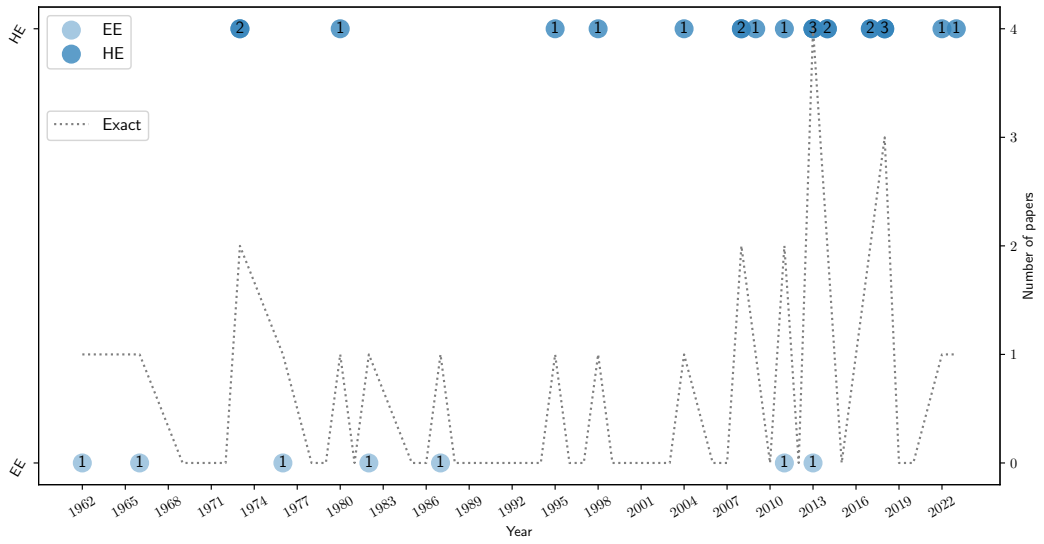

Figure 3. Published exact solvers per year.

4.3. Heuristic Approaches

Out of all considered literature items, the largest amount of papers introduced or analyzed heuristic approaches to the $P||C_{max}$ problem as can be seen in Figure 2. This is expected because the problem is $\mathcal{NP}$-hard and exact solvers require large amounts of resources. As aforementioned, in general, heuristics are not able to provide guarantees of optimality. However, there are some heuristic methods that provide solutions with provable quality and/or provable runtime estimation.

Definition 3 (α-Approximation). *An algorithm $\mathcal{A}$ is an α-approximation for an optimization problem Π if for each instance I of problem Π algorithm $\mathcal{A}$ computes a feasible solution to I and: $\mathcal{A}(I) \leq \alpha OPT_{\Pi}(I)$, if Π is a minimization problem and $\alpha > 1$; or $\mathcal{A}(I) \geq \alpha OPT_{\Pi}(I)$, if Π is a minimization problem and $\alpha < 1$.*

If α is a constant, then an α-approximation is also known as a constant AR. For $P||C_{max}$ if $n \to \infty$, it could be said that it has 2-approximation [127], 4/3-approximation [128], 13/11-approximation [129], 72/61-approximation [53], etc. Usually, AR is not a constant, as it depends on the size of the input data.

Heuristic approaches, also referred to in the literature as approximate algorithms, may be divided into three groups. One group contains algorithms that construct a solution, the other is composed of algorithms improving given solution as categorized in [8,130], while the algorithms from the third group provide solutions arbitrarily close to the optimal and are recognized in [19] as follows:

- Constructive Heuristics (CH);
- Improvement Heuristics (IH);
- Polynomial Time Approximation Schemes (PTAS).

The CH algorithms quickly build a feasible solution. Some of those algorithms might provide also the AR. The goal of IH algorithms is to enhance the quality of a given solution and, similar to CH, some of them may provide AR. PTAS algorithms take the solution quality as an input argument and provide a solution of a given quality in polynomial time.

4.3.1. Constructive Heuristics

The CH algorithms are able to construct feasible solutions for a given instance very fast. Their known characteristics may include AR and WTC, which give a solid performance prediction in terms of solution quality and execution time, respectively. CHs are used

both independently and integrated in more complex optimization algorithms. With their execution time being often smaller than most lower-bounding procedures, they are usually executed just after the trivial lower bound (L_2). That is, CH algorithms are integrated in more complex optimization algorithms with the goal of providing a good upper bound for more complex lower-bounding procedures, improvement heuristics, exact solvers, and to solve easy instances fast (Section 4.1).

The CH algorithms recognized in our SLR study are summarized in Table 4. They are based on three identified techniques: List Scheduling (LS), Binary Search, and Decomposition, which are explained in the remainder of this section.

LS technique consists of two main steps. The first one is to arrange jobs into a priority list according to some predefined criterion. The second step performs actual assignment/scheduling of one-by-one jobs from the list to the most appropriate machine as it is defined by some scheduling rule. Many CH algorithms use the LS technique with different job ordering criteria and different scheduling rules.

The first $P||C_{max}$ optimization algorithm from the CH group was developed in 1966 [127]. In the literature, it is often referred to as LS because it uses the LS technique. However, we call it Simple List Scheduling (SLS) as the ordering criterion of jobs is not specified. The scheduling rule utilized in SLS assumes scheduling the current job on the least loaded machine, i.e., on the machine that is the earliest to start its execution. Therefore, this scheduling rule is often referred to as the Earliest Start (ES) scheduling rule.

Table 4. Constructive heuristics.

Name	Reference	Known Characteristics	Compared with
SLS	[127] 1966	AR: $2-\frac{1}{m}$; WTC: $\mathcal{O}(n \log m)$	N/A
LPT, KK	[128] 1969	AR: $\frac{4}{3}-\frac{1}{3m}$ ($\frac{7}{6}$ for $m=2$); WTC: $\mathcal{O}(n \log(nm))$, AR: $1+\frac{1-1/m}{1+\lfloor r/m \rfloor}$; WTC: $\mathcal{NP}$	N/A
MF′	[131] 1972	AR: $\frac{5}{4}+\frac{5}{4}2^{1-k}$; WTC: $\mathcal{O}(n \log n + kn \log m)$	N/A
MF	[129] 1978	AR: $\frac{13}{11}+2^{-k}$; WTC: $\mathcal{O}(n \log n + kn \log m)$	[128,131]
MAAAT	[112] 1980	WTC: $\mathcal{O}(n \log n)$	N/A
LDM	[53] 1982	AR: $\left[\frac{4}{3}-\frac{1}{3(m-1)}, \frac{4}{3}-\frac{1}{3m}\right]$ ($\frac{7}{6}$ for $m=2$); WTC: $\mathcal{O}(n \log n + nm \log m)$	N/A
MF_i	[132] 1986	AR: $\frac{72}{61}+2^{-k}$ (tight for $m \geq 12$); WTC: $\mathcal{O}(n \log n + kn \log m)$	N/A
MF_e	[133] 1988	AR: $\frac{10}{9}$ for $m=2$; WTC: $\mathcal{O}(n \log n + kn \log m)$	[128,129]
COMBINE	[134] 1988	AR: $\frac{13}{11}+2^{-k}$ ($\frac{10}{9}$ for $m=2$); WTC: $\mathcal{O}(n \log n + kn \log m)$	[128,129]
P_1	[135] 1994	WTC: $\mathcal{O}(n)$	N/A
MS	[57] 1995	WTC: $\mathcal{O}(mn^2)$	N/A
RAS_2	[136] 1996	AR: $\frac{4}{3}-\frac{1}{3m}$; WTC: $\mathcal{O}(n \log(nm))$	[128,129]
LISTFIT	[137] 2001	AR: $\frac{13}{11}+2^{-k}$; WTC: $\mathcal{O}(n^2 \log n + kn^2 \log m)$	[128,129,134]
FGH, DGH	[130] 2001	AR: $\frac{13}{11}+2^{-k}$; WTC: $\mathcal{O}(n \log n + kmn)$	[128,129,136]
AP	[138] 2002	WTC: $\mathcal{O}(n \log(nm))$	[128–130]
H_1	[139] 2004	WTC: $\mathcal{O}(nm \log(\frac{1}{\varepsilon}+\frac{1}{\varepsilon^2}))$	N/A
SS, LPT_R	[95] 2006	WTC: $\mathcal{O}(nmS)$, AR: $2-\frac{1}{m}$; WTC: $\mathcal{O}(n \log(nm))$	[57,128]
MPS	[140] 2007	AR: $1+\frac{m-1}{mz}$; WTC: $\mathcal{O}(n \log n + nm)$	[128]
PSC	[141] 2008	WTC: $\mathcal{O}(n \log n)$	[128]
PSC_i	[142] 2009	WTC: $\mathcal{O}(n \log n)$	[128]
SPS	[143] 2010	AR: $1+\frac{m-1}{mz}$; WTC: $\mathcal{O}(n \log n)$	[128]
DJMS	[144] 2015	AR: $\frac{13}{11}+2^{-k}$; WTC: $\mathcal{O}(n \log n + mkn \log m)$	[128,129,134,137]
PSMF	[145] 2015	AR: $\frac{13}{11}+2^{-k}$; WTC: $\mathcal{O}(kn^2)$	[128,129,134,137]
SLACK	[146] 2020	AR: $\frac{4}{3}-\frac{1}{3(m-1)}$ ($\frac{9}{8}$ for $m=2$); WTC: $\mathcal{O}(n \log(nm)$	[53,128,134]

z—number of initial partitions. k—number of steps. r—number of optimally scheduled longest jobs. S—target sum of job durations. ε—target precision.

The most popular LS-based algorithm is LPT [128], because of its simplicity, speed, and very good AR and WTC. The more appropriate name for this algorithm would be LPT+ES (as it is suggested in [147]). The LPT part of the name refers to the longest processing time job ordering criterion, while ES is used as a scheduling rule. However, the LPT+ES algorithm is most commonly referred to as the LPT algorithm only and we keep this notation. To distinguish between the two LPT meanings, we always write the LPT ordering criterion and LPT algorithm when appropriate. In the same study [128], Kleitman and Knuth's (KK) algorithm is presented. It starts with a partial schedule in which $r < n$ longest jobs are optimally scheduled and then arbitrarily schedules the remaining jobs. It has been proven that, under the optimality of partial solution and for a large enough r, KK improves AR in comparison to the LPT algorithm.

Several algorithms leverage the LPT job ordering criterion. m-Assignment-at-a-time (MAAAT), proposed in [112], uses the LPT job ordering criterion and schedules jobs to machines in a zig-zag manner. Ordered jobs are partitioned into subsets of size m and jobs from each subset are scheduled to machines alternating between first-to-last and last-to-first order. Randomized LPT (LPT_R) [95] iteratively selects two unscheduled jobs with the longest processing times and then randomly assigns one out of this pair to the first available machine.

RAS_2 [136] heuristic is similar to the LPT algorithm, but instead of ES, Best Fit (BF) is applied to schedule the remaining jobs. That is, the unscheduled job with the longest processing time is assigned to the most heavily loaded machine such that L_1 bound is not exceeded. If such a machine does not exist, the job is assigned to the least loaded machine. Therefore, AR and WTC are the same as for the LPT algorithm. In [148], AR was proved to be the same as LPT, and we concluded that WTC is the same as LPT, too.

Fixed Gap Heuristics (FGH) and Dynamic Gap Heuristics (DGH) [130] are also based on the LS technique and have parametric and non-parametric variants. They use the LPT job ordering criterion and an alternation between ES and McNaughton Adjustment (McNA) [17,136] scheduling rules. According to McNA, the machine whose load, after the assignment of the current job, will be closest to the makespan lower bound (without exceeding it) is selected. If this is not possible, the job is assigned to the machine where its completion time would be minimal. The presented procedures work in iterations, using the best solution makespan and various lower bounds [130] for switching between the McNA and ES scheduling rules. Both of these procedures have significantly better AR and significantly larger WTC in comparison to the LPT algorithm.

Three constructive heuristics [138] are based on the following algorithm. The setup is to start with a jobs sorted in a non-increasing order of their processing times and machines divided in r subsets ($r \leq m$). The algorithm takes the first subset of machines and assigns the jobs according to the LPT rule, until completion time of the included machines is nearest to a fixed limit (e.g., some lower bound), without exceeding it. When there is no machine in the first set to which it is possible to assign a job, respecting the limit, the next subset of machines is added to the first one and the algorithm follows in the same manner. When the last subset of machines was added the algorithm follows the LPT rule, assigning the rest of the jobs without respecting given limit. AP heuristics (AP_{10}, AP_{ni}, and AP_i) are derived based on the described algorithm depending on the way of forming the machine subsets. We concluded that heuristics can be implemented in WTC $\mathcal{O}(n \log(nm))$.

SLACK [146] is a recent modification of the LPT algorithm. It organizes jobs into groups (slacks) of m elements (complemented with zeros) and sorts them in a non-increasing order according to the difference between their longest and shortest jobs. In this way, a new priority list of jobs is obtained and jobs are scheduled one-by-one following the ES

scheduling rule. The AR of the SLACK algorithm is improved in comparison to the LPT algorithm, without significantly increasing the computation time.

P_1 from the 3-PHASE [135] algorithm is also based on LS and does not require sorting. It starts with identifying minimum and maximum job processing time and dividing the interval between these two values into a predefined number c of equal or almost equal sub-intervals. In addition, an array of c elements is introduced and initialized with zeros. Each element contains the index of the machine where the previous job from the corresponding sub-interval is scheduled. Then, the algorithm takes one by one job from the unsorted list and schedules it to the machines in a round round-robin manner. That is, if the previous job from the same sub-interval is scheduled to the machine i, the current one is assigned to machine $i \pmod m + 1$. AR was not provided in the study, but the scheduling can be performed in linear time. With well-chosen parameter c, a balanced schedule can be obtained.

MULTIFIT algorithm (MF), developed in 1978 [129], is based on the binary search technique, firstly used in [131] for similar but weaker and less popular MF′ algorithm. At the beginning of the MF algorithm lower (C_L) and upper (C_U) bounds on the search interval have to be set. MF iteratively performs a binary search in the interval $[C_L(i), C_U(i)]$, where, $C_L(1) = L_1$ and $C_U(1) = \max\{2L_1, p_{max}\}$. The main step of the binary search involves the execution of the First Fit Decreasing (FFD) algorithm [77,149] with the bin capacity limit $C = (C_L(i) + C_U(i))/2$ to construct the feasible solution for the BPP. The maximum number of iterations (steps) k is an input parameter of the MF algorithm. Recommended value for k is to be not smaller than seven, and seven is the default value if some another value is not mentioned. As AR depends on this input parameter, MF has a better AR for relatively small values of k, while having a higher WTC in comparison to the LPT algorithm. MF_e [133] and MF_i [132] are the modifications of MF.

The MPS [140] algorithm is based on the decomposition technique and it iteratively combines partial solutions (PS) of a given instance. The set of jobs is divided into a given number (z) of partitions. Each partition contains m disjunctive subsets of jobs, representing an initial PS, which are combined until a complete solution is obtained. The authors derived the properties that each PS should satisfy such that the resulting complete solution has AR $\frac{z+1}{z} - \frac{1}{mz}$. Although MPS' AR is strongly related to the number of initial partitions, it has been demonstrated that its values are competitive with the LPT's AR. A similar idea is used in PSC [141], and its improved variant PSC_i [142]; however, the AR for these two algorithms cannot be easily estimated. The newest algorithm in this group, SPS proposed in [143], has the same AR and lower WTC with respect to MPS.

There are several CH algorithms that are based on a combination of the identified techniques. The multi-Subset (MS) algorithm [57] uses the lower bound of L_3 and an efficient G^2 heuristic [150] for SSP. In the first phase, using G^2 it construcs at schedule the closest possible to L_3 for machines one by one, using currently unassigned jobs. In the second phase, if some jobs are still unscheduled, it is proved (assuming that G^2 has found an optimal solution) that their number is smaller then m. The optimal way to schedule them is to assign the longest unscheduled job to the less busy machine and so on. AR for this heuristic has not been derived. A similar CH algorithm to MS was developed for HI IH [139] and it is named H_1. In this variant the longest yet unassigned task is assigned to the current processor. Then, a subset of the yet unassigned tasks such that the sum of their processing times is as close as possible to a given limit to the makespan will be assigned to the same processor. The MTSS(3) PTAS [151] is used in the second step. The remaining unassigned tasks are considered one by one in non-increasing order of their processing times and assigned to the processor with the smallest load. SS algorithm starts by finding a subset of jobs whose total processing time is minimal but not less than

the value of a L_2, using the SSP solver. These jobs will be assigned to a first machine. Then, the algorithm computes a lower bound of the instance, which is defined by one machine less and the rest of the jobs. Again, a SSP is solved in order to determine an optimal subset of jobs that will be assigned to a second machine, and so on. In their implementation, authors used pseudo-polynomial time using the dynamic programming algorithm developed by Pisinger [152]. The COMBINE algorithm [134] is a combination of LPT and MF algorithms. It first applies the LPT algorithm and then invokes the MF algorithm with $C_L(1) = \max\{L_1, C_{max}^{LPT}/AR^{LPT}\}$ and $C_U(1) = C_{max}^{LPT}$. The COMBINE algorithm has the same AR as MF and provides competitive results that correspond to the better among LPT and MF algorithms.

The Different Job and Machine Sets (DJMS) algorithm [144] applies to most m times LPT algorithm followed by MF algorithm on various subproblems of the considered $P||C_{max}$. Unlike other LS-based algorithms, DJMS decomposes both job and machine lists.

The LISTFIT [137] algorithm combines MF with two LS techniques involving LPT and Shortest Processing Time (SPT) ordering criteria. The main idea of the LISTFIT algorithm is to create a number of job lists to be explored by the MULTFIT algorithm. In more detail, the set of jobs is iteratively partitioned into two subsets that are ordered according to all combinations of SPT and LPT criteria and used for generating various job lists. In such a way, $4n$ different lists are explored in the MULTFIT algorithm. The best obtained solution is returned as the final one. LISTFIT's AR corresponds to the MF algorithm, while WTC is significantly larger.

The PSMF algorithm [145] is a recent combination of modified MPS and MF algorithms that has the same AR as MF but worse WTC than both MPS and MF.

The largest Differencing Method (LDM) also known as the Karmarkar–Karp algorithm [53] is based on the idea of how to construct a solution hierarchically in n iterations. In the beginning, jobs are sorted in the non-decreasing order of their processing times, and n partial solutions are created such that j-th partial solution consists of $m-1$ empty machines, while machine m is assigned job j only. In each of the remaining $n-1$ iterations, two partial solutions are replaced with the result of their combination performed in the following way: First, two partial solutions with the maximum difference between the loads of maximally and minimally loaded machines are identified. These two solutions are combined into a new partial solution by joining the load of the minimally loaded machine from the first with the load of the maximally loaded machine in the second partial solution. The second smallest load of the first partial solution is joined with the second largest load of the second partial solution, and so on. In the end, a unique complete solution is obtained, whose AR value is estimated to be between ARs of SLACK and LPT. However, it has a larger WTC than SLACK.

4.3.2. Improvement Heuristics

The IH algorithms start from an existing feasible solution (however it is provided) and iteratively apply different modifications (transformations) trying to improve its makespan. The most frequently applied modifications are move and swap (exchange). The move modification, takes one or more jobs from one machine and schedules it/them to some other machine. In its most general case, swap modification selects two machines, takes subsets of jobs assigned to each of them and exchanges their positions. The simplest and most common swap variant is the binary swap that exchanges positions (machines) for a pair of jobs scheduled on different machines. We use term *interchange* to denote any modification of some feasible solution. One of the important properties of the IH algorithms is that they can be provided by a stopping criterion to limit their execution. The best found

solution is returned as the final result. Table 5 presents all improvement heuristics known to us, with information about other improvement heuristics they were compared with.

IH, proposed in 1978 [153], involves human help (interaction) in some steps of execution. Such IH approach is not practical, especially for large instances, and therefore, it was not further developed.

IC (0/1-INTERCHANGE), proposed in [154], is one of the simplest IH algorithms. It starts from a randomly generated feasible solution and iteratively performs the following steps. It sorts machines in the non-increasing order of their loads. It then calculates the difference between the most and the least heavily loaded machines. On the most heavily loaded machine, it finds the job with the processing time smaller than the calculated difference and moves it to the least heavily loaded machine. The above-mentioned steps are repeated until no such job can be found. Two improved versions of the IC algorithm (ICI and ICII) were developed in [155]. The final version of ICII differs from the original algorithm in the initial solution generation procedure. The longest $2m$ jobs are scheduled using the LPT algorithm, while the remaining jobs are assigned to the machines randomly. In this way, the same AR as the LPT algorithm is obtained without increasing WTC.

A more sophisticated approach, named Knapsack-on-multiple-processors (KOMP) algorithm, was proposed by[112]. The initial solution could be generated by any CH and the authors proposed to use LPT, MF, or their own MAAAT constructive procedure. In the improvement phase, the initial solution is first modified by iterative single-job moves and then by binary swaps (two jobs exchange their machines). Next, the resulting solution is additionally improved by iteratively solving the Knapsack problem for a pair of machines containing the most heavily loaded one using KP solver presented in [156].

The EX (EXCHANGE) algorithm [157] combines the interchange principle with the decomposition. The authors analyzed in detail the case when $m = 2$ and proposed to swap and/or move subsets of jobs such that the current makespan is reduced and made closer to L_0. In the more general case, a pair of machines (one of them being the most heavily loaded) is selected and 2-machine transformation rules are applied until no improvement can be made, iteratively. In [158], this algorithm is named EXCHANGE, and its AR is calculated.

The 3-PHASE algorithm [135] is composed of three procedures (phases). The first procedure is fast CH P_1. The second procedure repeatedly tries to move a job from the most heavily loaded machine to the least loaded machine in order to reduce the makespan. The third procedure tries to improve makespan by exchanging one job between the maximally loaded machine and some other machine.

Table 5. Improvement heuristics.

Name	Reference	Known Characteristics	Compared with
N/A	[153] 1978	Interactive	N/A
IC	[154] 1979	Interchange; AR: $2-\frac{2}{m+1}$; WTC: $\mathcal{O}(n \log m)$	N/A
KOMP	[112] 1980	KP + Interchange; WTC: $\mathcal{O}(nmS)$	N/A
EX	[157] 1981	Interchange + Decomposition; AR: $\frac{4}{3}-\frac{2}{3m}$; WTC: $\mathcal{O}(n^2mS)$	N/A
ICI, ICII	[155] 1982	Interchange; AR: $\frac{3}{2}-\frac{1}{2m}$; WTC: $\mathcal{O}(n \log m)$, Interchange; AR: $\frac{4}{3}-\frac{1}{3m}$; WTC: $\mathcal{O}(n \log m)$	N/A
3-PHASE	[135] 1994	Interchange + Decomposition; AR: 2	[154,155]
X-TMO	[159] 1995	SSP; AR: $\frac{13}{11}+2^{-k}$; WTC: $\mathcal{NP}$	N/A
RAS_1	[136] 1996	Interchange; WTC: $\mathcal{O}(n \log(nm))$	N/A
PI	[160] 1998	Interchange; WTC: $\mathcal{O}(n \log(nm))$	N/A
CP_{99}	[17] 1999	Cutting plane + ILP + Polyhedral theory	N/A
LPT+, MF+	[161] 2002	Interchange; AR: $\frac{4}{3}-\frac{1}{3m}$; WTC: $\mathcal{O}(nmS \log n)$, Interchange; AR: $\frac{13}{11}+2^{-k}$; WTC: $\mathcal{O}(nmS \log n)$	N/A
ME	[162,163] 2004	Interchange + Graph	[135]
HI	[139] 2004	BPP+TS; AR: $\frac{5}{4}+2^{-k}$	[135,163]
MSS	[95] 2006	SSP; AR: $2-\frac{1}{m}$; WTC: $\mathcal{O}(nmkS^2)$	[57,139]
MSK	[14] 2008	KP; AR: $2-\frac{1}{m}$; WTC: $\mathcal{O}(n^2mkS^2)$	[57,95,139]
CA	[164] 2011	Partial	[139,163]
PSMF+	[145] 2015	Interchange + Partial; AR: $\frac{13}{11}+2^{-k}$;	[139,163]
KL_h	[122] 2018	Interchange + BPP	[139]
MMIPMH	[165] 2019	Hopfild; AR: $\frac{4}{3}-\frac{1}{3m}$	N/A
SLACK+	[146] 2020	Interchange; AR: $\frac{4}{3}-\frac{1}{3(m-1)}$	N/A
N/A	[166] 2022	Interchange + KP	N/A
DIST	[12] 2023	Decomposition + MKP; AR: $\frac{4}{3}-\frac{1}{3(m-1)}$; WTC: $\mathcal{NP}$	N/A

X-TMO [159] uses the new TMO SPP solver to optimally solve a case that includes the most heavily loaded and the least loaded machine. When the solution cannot be improved anymore using these two machines, the procedure uses the second least loaded machine. Then, it continues choosing the next least loaded machine while there exists a chance for improvement.

CP_{99} [17] is an approximation algorithm that leverages linear programming formulations combined with a cutting planes method. This approach involves iterative calculations of lower bounds through successive linear programming relaxations.

RAS_1 [136] and PI [160] algorithms are similar to X-TMO, but they do not solve two-machine problems exactly. Instead, they try to find the best two jobs on two machines for exchange. The presented WTC for RAS_1 in [136] is $\mathcal{O}(n \log n + nm \log m)$; however, we concluded that (using efficient data structures) both algorithms can be implemented in WTC $\mathcal{O}(n \log(nm))$.

LPT+ and MF+ [161] start with schedules constructed with LPT and MF CHs. A local search heuristic is applied to refine the solutions generated by constructive heuristics. The neighborhood is created by identifying pairs of jobs assigned to processors with varying loads, ensuring that one of the processors is the most heavily loaded. The local search involves evaluating these job pairs and accepting changes that decrease the load disparity between the two processors. After each adjustment, the neighborhood is redefined, and the search process begins again. The search ends when all job pairs in the neighborhood have been explored without achieving any improvement, indicating that a local optimum has been reached.

A wide family of IH solvers for $P||C_{max}$ has been proposed in [162,163]. The authors showed that, among them, 1-SPT, K-SPT, 1-BPT, and K-BPT exhibit the best performance. They are based on a graph representation of the problem and multiple exchanges of jobs between machines. The best result of the mentioned four heuristic is denoted as Multi-Exchange (ME).

Hybrid Improvement (HI) [139] starts with LPT and H_1 CHs, and ε-DUAL PTAS for $\varepsilon = \frac{1}{5}$ which also produces L_{HS} and gives AR of $\frac{5}{4} + 2^{-k}$. Additionally, lower bounds $L3$ and L_θ are used. Then, HI applies the binary search that explores duality between $P||C_{max}$ and BPP where for every new target value of makespan new starting solutions are constructed and improved with TS.

Multistart Subset Sum (MSS) [95] is an IH algorithm which tries to iteratively improve the solution by equalizing the load of all pairs of machines with the most heavily loaded one and using an exact SSP solver. The Multistart Knapsack (MSK) algorithm [14] works in the same way, but always puts a minimal number of jobs on a machine with a smaller load. This strategy increases the chance for future improvements. To achieve this, instead of an exact SSP solver, it uses an exact KP solver.

The Composite Algorithm (CA) [164] starts from a family of initial partial solutions and combines these partial solutions (similarly to MPS [140]) until a feasible solution is generated. In the second phase, the CA applies local search procedures on partial solutions trying to generate a feasible solution with an improved makespan compared to the initial feasible solution. In total, five local search procedures are defined, one exploring the single-job moves, while four of them are based on the job swaps. In two of the swap-based neighborhoods, it is allowed to exchange a single job with a subset of jobs (referred to as composite jobs) from another machine. In addition, two neighborhoods allow accepting the solutions of the same quality as the initial one, as long as these moves increase the chances of improving the succeeding neighborhoods (similarly to MPS).

The PSMF+ algorithm [145], is obtained by inserting a 2-exchange procedure in PSMF CH after CUPS procedure and after MTMF procedure. It swaps two jobs from different machines until an advantageous exchange has been identified.

The KL_h algorithm [122] is an initial phase of the KL HE algorithm. It consists of the constructive approximation algorithms, the local searches for improvements for every constructed solution, the lower bounds, and the primal heuristic at the root node of the B&P tree for the makespan value is equal to the best lower bound. The main component is the primal heuristic with task to find a feasible solution for a BPP instance at the root node of HE algorithm.

The MMIPMH method [165] is inspired by balancing the Hopfild neural network with binary decision variables from the assignment ILP model for $P||C_{max}$. In practice, it iteratively applies the LPT algorithm on selected partial solutions.

SLACK+ [146] applies a one-job swap neighbor search (NS) on a solution obtained with SLACK constructive heuristic. Swaps are performed between the most heavily loaded and some other machine.

In [166], a wide set of solution modification procedures that perform local improvements of an initial $P||C_{max}$ solution is presented. In total, 35 solution modifications are proposed and, when combined with two CH algorithms to obtain an initial solution, they produce 70 simple IHs. Solution modifications are composed of job moves and swaps and are divided into two classes depending on the selection strategies for both jobs and machines. The main goal of these transformations is to obtain a makespan of the schedule as close as possible to L_0 lower bound. Transformations from the first class are composed of the following steps: The most heavily loaded machine (called source) is identified, a job from it is selected, as well as the destination machine where this job should be moved. Optionally, the load imbalance may be improved by moving some small-sized jobs from the destination machine to the source one in a deterministic way. There are three different ways to select the job to be moved and five possibilities to select the destination machine. Transformations from the second class combine (mix) all jobs from the source and destination machines (identified in the same ways as for the first class transformations) and schedule

them by the KP algorithm such that the load imbalance is minimized. The complexity of all transformations is analyzed having in mind the efficient data structures that have been used.

Decomposition-based Iterative Stochastic Transformation (DIST) [12] is IH based on binary searching for optimal makespan and exact solving subproblems of increasing size using the MKP exact solver. DIST explores the relationship between $P||C_{max}$, BPP, and Multiple Subset Sum Problem (MSSP) end, examining efficient algorithms for BPP and MSSP to provide high-quality solutions of $P||C_{max}$. The main step of DIST explores a stochastic partitioning strategy to create subproblems of the original problem that are treated as instances of MSSP and solved by the efficient exact solver based on MKP within the predefined time limit. At the beginning of its execution, DIST starts by random partitions that create smaller subproblems with a larger potential to yield improvements of the current solution. In the case were the improvement has not been achieved, the algorithm creates and solves larger subproblems. In such a way, DIST performs nondeterministic transformations of the current solution, and therefore, it could be considered as a stochastic search algorithm. Due to the fact that it applies the time-limited exact solvers to the created subproblems, it belongs to the class of heuristic algorithms. However, DIST exhibits a very good performance within a short execution time, and becomes an exact solver for given big enough time limits.

4.3.3. Polynomial Time Approximation Schemes

The majority of constructive heuristics provide good approximations for the optimization of $P||C_{max}$. However, $P||C_{max}$ and many other optimization problems can be approximated to any precision. More precisely, there exists a special group of algorithms providing nearly optimal solutions that may be possible in polynomial time. They are known as Polynomial Time Approximation Schemes or PTAS algorithms [167]. In this section, we explain the characteristics of PTAS and review studies that introduced various PTAS algorithms.

Definition 4 (Approximation Scheme). *An approximation scheme for an optimization problem* Π *is a family of algorithms* $\mathcal{A}_\varepsilon$ *defined as follows. For a given approximation accuracy parameter* $\varepsilon > 0$ *and an instance* I *of problem* Π*, algorithm* $\mathcal{A}_\varepsilon$ *computes a feasible solution of* I *whose objective value differs from* $OPT_\Pi(I)$ *by at most* εOPT_Π*, i.e.,* $|\mathcal{A}_\varepsilon(I) - OPT_\Pi(I)| \leq \varepsilon OPT_\Pi(I)$.

Actually, PTAS is defined as an algorithm that, for any given $\varepsilon > 0$, produces a feasible solution within a factor of $(1+\varepsilon)$ of the optimal solution. The running time of the algorithm should be polynomial with respect to the input size and inversely proportional to ε. For example, if an algorithm is a $(1+\varepsilon)$-approximation PTAS for a certain problem, it means that the solution it provides is guaranteed to be within a factor of $(1+\varepsilon)$ times the optimal solution. As ε approaches zero, the approximation factor approaches one, indicating a solution that is arbitrarily close to the optimal.

The parameter ε controls the distance to the optimal objective value, i.e., smaller values of ε result in better solutions (closer to optimum). It is important not to confuse approximation schemes with simple heuristic algorithms, as the later may not provide certain approximation guarantees. Selecting the appropriate value for ε involves a trade-off. A smaller ε generally leads to a more accurate approximation but may result in longer running times. The choice of ε depends on the specific requirements of the problem at hand and the balance between solution quality and computational efficiency that is acceptable for the application. Researchers and practitioners in general cases often experiment with different values of ε to find a suitable compromise for their particular use case.

Let us denote by $|I|$ the size of instance I, and let $f : \mathbb{R} \to \mathbb{R}$ be a polynomial function. Depending on the running time, the following types of approximation schemes

are summarized in Table 6. A quasipolynomial-time approximation scheme (QPTAS) if the corresponding algorithms run in time $|I|^{f(1/\varepsilon)log^{\mathcal{O}(1)}|I|}$. A polynomial-time approximation scheme (PTAS) if they run in time $|I|^{f(1/\varepsilon)}$. An efficient polynomial-time approximation scheme (EPTAS) if they run in time $f(1/\varepsilon)|I|^{\mathcal{O}(1)}$. A fully polynomial-time approximation scheme (FPTAS) if they run in time $(|I|/\varepsilon)^{\mathcal{O}(1)}$, which is polynomial in $|I|$ as well as in $1/\varepsilon$.

Table 6. Approximation schemes.

Approximation Scheme	QPTAS	PTAS	EPTAS	FPTAS
WTC	$\|I\|^{f(1/\varepsilon)log^{\mathcal{O}(1)}\|I\|}$	$\|I\|^{f(1/\varepsilon)}$	$f(1/\varepsilon)\|I\|^{\mathcal{O}(1)}$	$(\|I\|/\varepsilon)^{\mathcal{O}(1)}$

In [102], the authors proposed an FPTAS for the $P_m||C_{max}$ problem, i.e., when the number of machines m is fixed. However, in general, the problem is strongly $\mathcal{NP}$-hard, so there is no FPTAS for $P||C_{max}$ unless $\mathcal{P} = \mathcal{NP}$. PTAS approach is not practical for $P||C_{max}$ because, even for large values of ε, the execution time is unacceptably long for large-size instances. Therefore, PTAS algorithms are usually studied from a theoretical aspect. WTC that includes ε are common comparison measures for this type of optimization algorithms.

The first PTAS for $P||C_{max}$ is ε-DUAL and it was presented in 1985 [54,168], and improved by [58,169–171]. For $(1+\varepsilon)$-approximate solutions, the fastest known PTAS for $P||C_{max}$ has a running time of $2^{\mathcal{O}((1/\varepsilon)\log(1/\varepsilon)\log\log(1/\varepsilon))} + \mathcal{O}(n)$ [175]. [107] shows that, assuming the exponential time hypothesis (ETH), for $\varepsilon > 0$ it cannot exist PTAS that yields $(1+\varepsilon)$-approximate solutions with running time $2^{(1/\varepsilon)^{1-\delta}} + poly(n)$ for any $\delta > 0$. Table 7 shows the WTC of all known PTAS optimization algorithms for $P||C_{max}$. Among them, only ε-DUAL and BDJR [174] have been implemented. Lower bounds for approximation schemes were studied by [64].

Table 7. Polynomial time approximation schemes.

Name	Reference	Known Characteristics	Compared with
ε-DUAL	[54,168] 1985	WTC: $n^{\mathcal{O}((1/\varepsilon)^2\log(1/\varepsilon))}$	N/A
N/A	[169] 1989	WTC: $n^{\mathcal{O}((1/\varepsilon)\log^2(1/\varepsilon))}$	N/A
N/A	[58,170] 1998	WTC: $2^{\mathcal{O}((1/\varepsilon)^{poly(1/\varepsilon)})} + \mathcal{O}(n\log n)$	N/A
N/A	[171] 2010	WTC: $2^{\mathcal{O}((1/\varepsilon^2)\log^3(1/\varepsilon))} + \mathcal{O}(n\log n)$	N/A
JR	[172] 2019	WTC: $2^{\mathcal{O}((1/\varepsilon)\log^2(1/\varepsilon))} + \mathcal{O}(n)$	N/A
N/A	[173] 2020	WTC: $2^{\mathcal{O}((1/\varepsilon)\log^4(1/\varepsilon))} + poly(n)$	N/A
BDJR	[174] 2022	WTC: $2^{\mathcal{O}((1/\varepsilon)\log(1/\varepsilon)\log\log(1/\varepsilon))}\log n + \mathcal{O}(n)$	N/A
N/A	[175] 2023	WTC: $2^{\mathcal{O}((1/\varepsilon)\log(1/\varepsilon)\log\log(1/\varepsilon))} + \mathcal{O}(n)$	N/A

4.3.4. Summary of Heuristic Approaches

One of the oldest constructive techniques, LPT proposed in 1966 [127], still presents the most popular heuristic for the $P||C_{max}$ problem due to its simplicity and effectiveness. Ever since its development, it has been used in other approaches to generate the initial solution. The different heuristic approaches developed from 1966 to today are summarized in Figure 4. The figure shows that it is easy to distinguish the development of the heuristic approaches in any decade and for any group of interest. There has been a steady development of improvement heuristics and that PTAS were introduced in 1980s and there has been renewed interest in that type of algorithms lately.

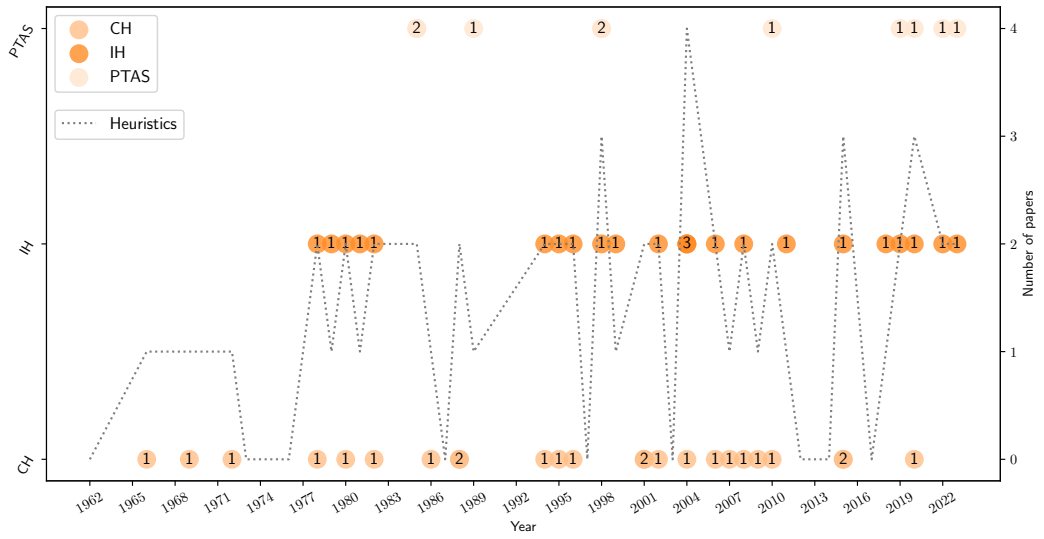

Figure 4. Published heuristics per year.

4.4. Metaheuristics

Metaheuristic (MH) algorithms [19] are high-level search strategies designed to find near-optimal solutions for optimization and search problems. These algorithms are not problem-specific but can be applied to a wide range of problems. They are particularly useful when dealing with complex problems for which finding an exact solution is impractical or impossible within a reasonable amount of time.

MH algorithms became popular in the late 1980s and their application to $P||C_{max}$ started at that time. We will use [19] classification and divide MH algorithms into Population-Based Metaheuristics (P-class) and Single Solution-Based Metaheuristics (S-class). P-class metaheuristics maintain a population of potential solutions throughout the search process. The algorithms maintain a population though its evolution over time, typically through mechanisms inspired by natural processes such as selection, mutation, and recombination. S-class metaheuristics focus on iteratively improving a single candidate solution. They generally perform local search around the current solution and move towards better solutions through various strategies. We are primarily interested in enumerating the approaches that were successfully applied to $P||C_{max}$.

All MH approaches are summarized in Table 8. A short review of studies of HS metaheuristics for $P||C_{max}$, without practical comparisons, is conducted in [176]. It could be seen that the S-class and P-class MH algorithms are equally represented. The majority of metaheuristics are performing transformations on the existing solutions, while only some sporadic approaches are based on the construction of new and potentially better solutions [177].

Table 8. Metaheuristics.

Name	Reference	Known Characteristics	Compared with
SA_{88}	[10] 1988	SA	N/A
HG	[178] 1994	TS	[10]
SA_{97}	[179] 1997	SA	N/A
TGR	[180] 1998	TS	[178]
GA_{99}, SA_{99}	[181] 1999	GA, SA	N/A
AIS, SA_{02}	[161] 2002	IBA	N/A
AntMPS	[7] 2003	ACO	N/A
SA_{06}	[182] 2006	SA	N/A
ILS_{06}	[183] 2006	ILS	N/A
HDNN	[184] 2006	ANN	N/A
$HDNN_i$	[185] 2007	ANN	N/A
TCNN, $TCNN_i$	[186] 2008	ANN	[184]
$DIMM_{SS}$	[114] 2008	SS	N/A
MC_{09}	[187] 2009	MC	N/A
MVNS, GA_{09}	[5] 2009	VNS, GA	N/A
BCO_{09}	[6] 2009	BCO	N/A
VNS_{09}	[188] 2009	VNS	[187]
$DPSO_{09}$, HDPSO	[189] 2009	PSO	[182]
DSHS	[190] 2010	HS	[182,189]
DHS_{11}, HDHS	[191] 2011	HS	[5,181,182]
RIVNS, HIVNS	[192] 2012	VNS	[5,181,182]
SA_{12}	[193] 2012	SA	[182]
SPPSO, $DPSO_{12}$, PSO_{spv}	[194] 2012	PSO	N/A
BCO_{12}	[177] 2012	BCO	N/A
DHS_{12}, BHS, DHS_{LS}	[195] 2012	HS, HS, HS+VNS	[182,189]
CSA	[196] 2015	CS	[182,189]
GES, GES+	[197] 2018	GES	[182]
ICSA	[198] 2018	CS	[182,189,195,196]
$RIVNS_1$, $HIVNS_1$, $RIVNS_2$, $HIVNS_2$	[199] 2018	VNS	[5,181,182,192]
GA_{19}, GWO_{19}	[200] 2019	GA, GWO	N/A

4.4.1. Summary of Metaheuristic Approaches

Figure 5 presents the distribution over time of papers describing the applications of different underlying metaheuristics. From the figure, it is easy to distinguish the development of the metaheuristic approaches in any decade and for any algorithm of interest. Up to ten years ago Simulated Annealing (SA) was the most popular metaheuristics [10,161,179,181,182,193]. The first metaheuristic algorithm for $P||C_{max}$ problem (MMBPP formulation) from 1988 [10] used SA. The Tabu Search (TS) was applied by [178,180]. The Genetic Algorithm (GA) was applied in [5,181,200]. The Harmony Search (HS) was applied most recently [190,191,195]. Other notable metaheuristics include the Iterated Local Search (ILS) [183], Genetics Algorithms (GA) [181], Gray Wolf Optimization (GWO) [200], Cuckoo Search (CS) [196,198], Immune-Based Approach (IBA) [161], Ant Colony Optimisation (ACO) [7], Scatter Search (SS) [114], Particle Swarm Optimization (PSO) [189,194], Monte Carlo (MC) [187], Bee Colony Optimization (BCO) [6,177], Grouping Evolutionary Strategy (GES) [197], and Variable Neighborhood Search (VNS) [5,188,192,199]. Deep learning methods, being successful in many different areas, might have a potential in applications to $P||C_{max}$. However, so far only methods that use Artificial Neural Networks (ANN) have been developed [184–186].

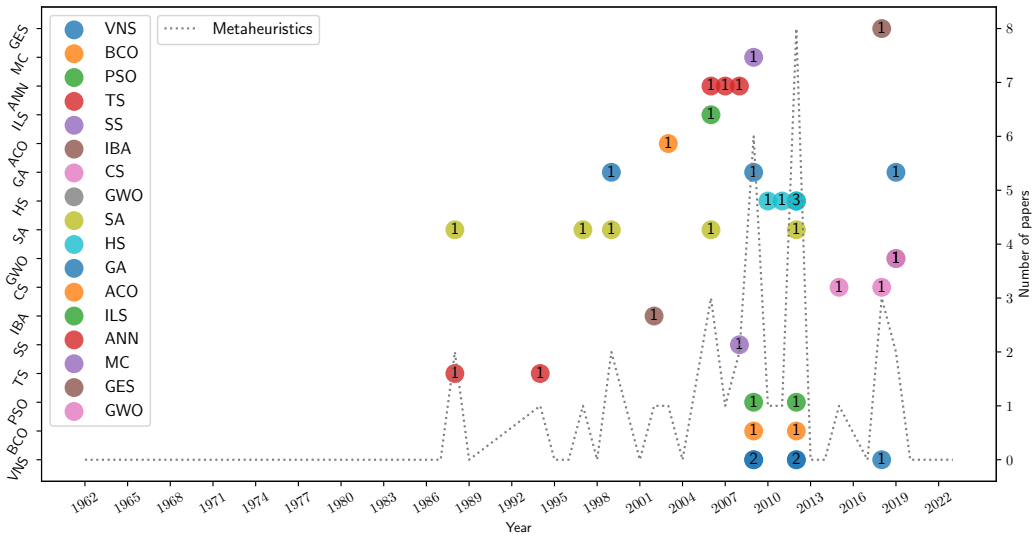

Figure 5. Published metaheuristics per year.

4.5. Parallel Optimization Algorithms

Parallelization of optimization methods is a very popular research avenue: however, its description is out of the scope of this study. Optimization algorithms from all three main categories can be parallelized. However, in our SLR, we found only four parallel algorithms for $P||C_{max}$. The parallel version of ε-DUAL PTAS proposed by [54] is published in [201], while [202] contains parallel versions of BCO_{12} MH [177], and in [203] parallelization of IRNP HE algorithm [116] is presented. BDJR PTAS is presented in [174] along with its parallel implementation.

4.6. Taxonomy of $P||C_{max}$ Optimization Algorithms

Figure 6 shows the taxonomy of $P||C_{max}$ optimization algorithms. Relationships between different types of algorithms are represented by the connecting lines and similar colors. HE solvers often use some heuristics and/or metaheuristic before B&B phase. Metaheuristics often use some heuristics for initial phase. IHs starts with some CH solution, and tries to improve it. Any type of algorithm can be more or less effectively parallelized.

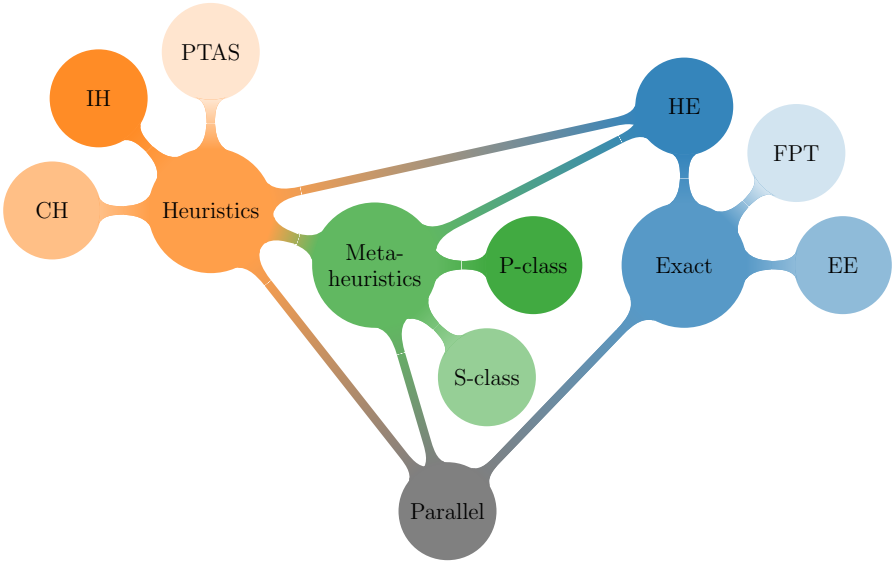

Figure 6. $P||C_{max}$ optimization algorithms taxonomy.

5. RQ2: Standardization of the Problem Instances

To evaluate the performance of any optimization algorithm, a representative set of benchmark test instances is needed. Unfortunately, our SLR shows that such a representative set does not exist for $P||C_{max}$. We found a number of various instance sets, but not all are available to wider audience. Therefore, our aim was to review the most commonly used sets, identify their strengths and weaknesses, classify them into various categories, and select a representative set for each category. Finally, we perform the standardization of identified groups of instances according to the criteria presented in the reminder of this section.

The most commonly used are the four groups of instances that we classify as follows:

- E instances;
- F instances;
- C instances;
- B instances.

Please note that the names for groups of instances do not systematically appear in the corresponding papers, and therefore, we provide detailed explanations. The description of each group of widely used instances includes a short history, main characteristics, and the list of papers exploring these instances in the experimental evaluation. At the end of this section, we provide a graph illustrating the utilization frequency of various instance groups in the relevant publications over time. From that graph it will be easy to distinguish the use of instance groups in any decade and for any group of interest. All standardized instances are now made available at the publicly accessible website https://gitlab.com/pcmax-problem/pcmax-instances. This repository contains either the generation code with the guidance how to use it or the set of instances ready for use.

In addition, we propose a uniform notation ${}^{N}_{M}I^{\frac{n}{m}}_{p}$. N and M representing maximal values for n and m, respectively, I denotes related set of instances, and D describes the distribution for generating job durations. $\widetilde{I}$ represents set of instance similar to I, and $\overline{I}$ represents subset of I. If different values for $\frac{n}{m}$ and/or D are exploited, they are all listed separated by comma. The correspondence between our notation and the instance names from the literature is described in the reminder of this section.

The main characteristics of instances from various groups are summarized in Tables 9–12. These tables are organized in the following way: The first column contains names of the subgroup of the presented group of instances (*class*), followed by the corresponding notation (that we introduced) in the second column (*notation*). The ratio between the number of jobs and the number of machines ($\frac{n}{m}$) is presented in the third column. The fourth column shows the number of machines (m), while the type of distribution and its corresponding parameters used to obtain values of processing times are given in the last column (D). For uniform distribution we are using notation $\mathcal{U}_{[a,b]}$, where $[a, b]$ represents interval for sampling values. For normal distribution, notation $\mathcal{N}_{\mu,\sigma}$, where μ represents mean and σ denotes standard deviation, is utilized. Each combination of values presented in columns three to five corresponds to one generation rule for instances (some authors also use the term *instance size*). An instance that has a solution for which $C^{*}_{max} = L_0$ is known as a *perfect instance*, and the corresponding solution is known as a *perfect schedule*, i.e., a *perfect partition*. In these types of schedules, the difference in the loads of any two machines is not greater than one.

At the end of this section, we also cover other instances that do not match any criteria defining the above mentioned groups. However, the introduced notation applies to those instances as well.

5.1. E Instances

The first widely used group of instances for the $P||C_{max}$ problem are what we will call E instances. E instances cover a huge range of parameter $\frac{n}{m}$. The first part of those instances, E_1 instances, was proposed in 1971 for testing of the developed heuristic in [204], the study that was never published. These instances were used also in [21,26]. Extension E_2 proposed in 1988 was used in [133,134] where normal distribution is letter replaced with uniform distribution, while E_3 and E_4 extensions appeared in 2001 [137]. The final version of the complete E group is shown in Table 9. The main idea behind E instances was to develop a framework that would serve for testing various algorithms used for $P||C_{max}$ problem.

This group has a variety of different values for $\frac{n}{m}$ parameter, which is generally preferred characteristic. All job's processing times are generated in a uniform way, with the smallest range $[10, 20]$ and the largest range being $[100, 800]$. However, values for m are too small to make these instances hard in any sense. Another big problem is the absence of information about random seeds used for their generation, as well as corresponding download repositories.

Today, E instances are obsolete for testing state-of-the-art optimization algorithms, although they are still in use for testing MHs [182,189,190,193–198,200], CHs [21,26,30,134,137,144,145], and measuring the performance of PTAS [174]. In all extracted studies, the authors generated E instances on their own with different random seeds. Thus, there is no guarantee that the obtained results could be compared fairly. To address this problem, in our repository we provide an open source generator for E instances and we propose to generate 50 instances per generation rule, using 1 as the seed value.

Table 9. E instances with 120 different generation rules.

class	*notation*	$\frac{n}{m}$	m	D
E_1	${}^{25}_{5}E_1{}^{2,3,5}_{\mathcal{U}_{[1,20]}\mathcal{U}_{[20,50]}}$	2, 3, 5	3, 4, 5	$\mathcal{U}_{[1,20]}, \mathcal{U}_{[20,50]}$
E_2	${}^{100}_{2}\overline{E_2}{}^{5,15,25,50}_{\mathcal{U}_{[100,800]}}$	5, 15, 25, 50	2	$\mathcal{U}_{[100,800]}$
	${}^{100}_{3}\overline{E_2}{}^{3.33,10,16.67,33.33}_{\mathcal{U}_{[100,800]}}$	3.33, 10, 16.67, 33.33	3	
	${}^{100}_{4}\overline{E_2}{}^{7.5,12.5,25}_{\mathcal{U}_{[100,800]}}$	7.5, 12.5, 25	4	
	${}^{100}_{6}\overline{E_2}{}^{5,8.33,16.67}_{\mathcal{U}_{[100,800]}}$	5, 8.33, 16.67	6	
	${}^{100}_{8}\overline{E_2}{}^{3.75,6.25,12.5}_{\mathcal{U}_{[100,800]}}$	3.75, 6.25, 12.5	8	
	${}^{100}_{10}\overline{E_2}{}^{3,5,10}_{\mathcal{U}_{[100,800]}}$	3, 5, 10	10	
E_3	${}^{17}_{3}\overline{E_3}{}^{3.33,3.67,4.33,4.67,5.33,5.67}_{\mathcal{U}_{[1,100]}\mathcal{U}_{[100,200]}\mathcal{U}_{[100,800]}}$	3.33, 3.67, 4.33, 4.67, 5.33, 5.67	3	$\mathcal{U}_{[1,100]}, \mathcal{U}_{[100,200]}, \mathcal{U}_{[100,800]}$
	${}^{27}_{5}\overline{E_3}{}^{3.2,3.4,4.2,4.4,5.2,5.4}_{\mathcal{U}_{[1,100]}\mathcal{U}_{[100,200]}\mathcal{U}_{[100,800]}}$	3.2, 3.4, 4.2, 4.4, 5.2, 5.4	5	
	${}^{42}_{8}\overline{E_3}{}^{3.12,3.25,4.12,4.25,5.12,5.25}_{\mathcal{U}_{[1,100]}\mathcal{U}_{[100,200]}\mathcal{U}_{[100,800]}}$	3.12, 3.25, 4.12, 4.25, 5.12, 5.25	8	
	${}^{52}_{10}\overline{E_3}{}^{3.1,3.2,4.1,4.2,5.1,5.2}_{\mathcal{U}_{[1,100]}\mathcal{U}_{[100,200]}\mathcal{U}_{[100,800]}}$	3.1, 3.2, 4.1, 4.2, 5.1, 5.2	10	
E_4	${}^{9}_{2}\overline{E_4}{}^{4.5}_{\mathcal{U}_{[1,20]}\mathcal{U}_{[20,50]}\mathcal{U}_{[50,100]}\mathcal{U}_{[100,200]}\mathcal{U}_{[100,800]}}$	4.5	2	$\mathcal{U}_{[1,20]}, \mathcal{U}_{[20,50]}, \mathcal{U}_{[50,100]}, \mathcal{U}_{[100,200]}, \mathcal{U}_{[100,800]}$
	${}^{10}_{3}\overline{E_4}{}^{3.33}_{\mathcal{U}_{[1,20]}\mathcal{U}_{[20,50]}\mathcal{U}_{[50,100]}\mathcal{U}_{[100,200]}\mathcal{U}_{[100,800]}}$	3.33	3	

5.2. F Instances

The next identified group of widely used instances is F, in relevant papers often called benchmark instances. In the first part of those instances (uniform), F_U, introduced in 1994 [135] for the purpose of testing 3-PHASE IH, is proposed by the authors. Ten years later, in 2004, the (non-uniform) extension, F_{NU}, of this group of instances is proposed

by [163] with the purpose of providing harder instances required by their ME IHs. *F* instances are presented in Table 10.

Table 10. *F* instances with 78 different generation rules.

class	*notation*	$\frac{n}{m}$	m	D
F_U	$^{10^3}_{5}\overline{F_U}^{2,10,20,100,200}_{\mathcal{U}_{[1,100]},\mathcal{U}_{[1,10^3]},\mathcal{U}_{[1,10^4]}}$	2, 10, 20, 100, 200	5	$\mathcal{U}_{[1,100]}, \mathcal{U}_{[1,10^3]}, \mathcal{U}_{[1,10^4]}$
	$^{10^3}_{10}\overline{F_U}^{5,10,50,100}_{\mathcal{U}_{[1,100]},\mathcal{U}_{[1,10^3]},\mathcal{U}_{[1,10^4]}}$	5, 10, 50, 100	10	
	$^{10^3}_{25}\overline{F_U}^{2,4,20,40}_{\mathcal{U}_{[1,100]},\mathcal{U}_{[1,10^3]},\mathcal{U}_{[1,10^4]}}$	2, 4, 20, 40	25	
F_{NU}	$^{10^3}_{5}\overline{F_{NU}}^{2,10,20,100,200}_{\xi_1,\xi_2,\xi_3}$	2, 10, 20, 100, 200	5	ξ_1, ξ_2, ξ_3
	$^{10^3}_{10}\overline{F_{NU}}^{5,10,50,100}_{\xi_1,\xi_2,\xi_3}$	5, 10, 50, 100	10	
	$^{10^3}_{25}\overline{F_{NU}}^{2,4,20,40}_{\xi_1,\xi_2,\xi_3}$	2, 4, 20, 40	25	

$$\xi_1 \sim \begin{cases} \mathcal{U}_{[89,100]}, & 98\% \text{ cases} \\ 1, & \text{otherwise} \end{cases}. \quad \xi_2 \sim \begin{cases} \mathcal{U}_{[899,10^3]}, & 98\% \text{ cases} \\ \mathcal{U}_{[1,19]}, & \text{otherwise} \end{cases}. \quad \xi_3 \sim \begin{cases} \mathcal{U}_{[8999,10^4]}, & 98\% \text{ cases} \\ \mathcal{U}_{[1,199]}, & \text{otherwise} \end{cases}.$$

F instances are used for testing all types of algorithms for $P||C_{max}$ problem. For HE solvers, this group of instances was used in [8,14,57,63,114,119,122], where the authors in [14,95,122,135,139,145,146,161–164] utilized them for IHs; CHs have been tested on these instances by [139,141–143,146], while [114,161] explored F instances to evaluate the performance of MHs. Even though links to the instances provided in some of the papers are not active anymore, all methods have been tested on exactly the same instances that we are providing in our repository. For each generation rule, there are ten generated instances.

With respect to standardization issues, *F* instances satisfy the majority of criteria and are accompanied with optimal solutions that can be retrieved from both the above mentioned papers and our repository.

5.3. C Instances

In numerous extracted studies, we found a few sets of instances that are generated with similar rules and thus, having similar characteristics. We refer to all these sets as *C* instances. The first set of *C* instances, based on the instance set for BPP problem [125], was proposed in 1995 for comparing exact solvers [57]. Since then, they have been used sporadically, while lately, these instances become very popular and are constantly upgraded. The *C* instances were divided into five distinctive classes with respect to the distributions used to generate the length of jobs. In these classes, D values follow distributions: $\mathcal{U}_{[1,100]}$, $\mathcal{U}_{[20,100]}$, $\mathcal{U}_{[50,100]}$, $\mathcal{N}_{100,20}$, and $\mathcal{N}_{100,50}$, respectively. These instances are initially generated with a relatively big $\frac{n}{m}$ ratio, where m takes values between 2 and 15 and values for n belong to the $[10, 10^4]$ interval. In [95], there are similarly generated instances having a smaller $\frac{n}{m}$ factor with an aim to produce a harder set of instances. More precisely, the authors used $\frac{n}{m} = 2.5$ and $D \in \mathcal{U}_{[\frac{n}{5}, \frac{n}{2}]}$, for each $m \in \{8, 12, 16, 20, 24, 28, 32, 36, 40, 60, 80\}$. This was the first time that parameters of D did not take constant values, instead they were dependent on the input data defining each particular instance. Inspired by [95], authors in [8] added two new distribution for generating lengths of jobs ($\mathcal{U}_{[n,4n]}$ and $\mathcal{N}_{4n,n}$), with additional values for $\frac{n}{m} \in \{2, 2.25, 2.75\}$. Later, $\frac{n}{m} = 3$ is added in [123] where values of m have been used from similar range as in [95]. The classes of *C* instances used in [123], with included (m, n) pairs $\{(8, 18), (6, 18)\}$, and $\frac{n}{m}$ values $\{4, 4.5, 5, 6, 9, 10, 11\}$, are summarized in Table 11 and considered as the final version. For all instances, ten times generated per generation rule, from this final version are provided in our repository. Different versions of *C* instances were used for testing HE solvers [8,12,14,57,63,95,123,124].

Table 11. *C* instances

class	*notation*	$\frac{n}{m}$	*m*	*D*
C_p	${}^{200}_{100}\overline{C}^{2}_{D}$	2	10, 20, 30, 40, 50, 60, 70, 80, 90, 100	$\mathcal{U}_{[1,100]}$, $\mathcal{U}_{[20,100]}$, $\mathcal{U}_{[50,100]}$, $\mathcal{N}_{100,20}$, $\mathcal{N}_{100,50}$, $\mathcal{U}_{[n,4n]}$, $\mathcal{N}_{4n,n}$
	${}^{198}_{88}\overline{C}^{2.25}_{D}$	2.25	8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88	
	${}^{200}_{80}\overline{C}^{2.5}_{D}$	2.5	8, 16, 24, 32, 40, 48, 56, 64, 72, 80	
	${}^{220}_{80}\overline{C}^{2.75}_{D}$	2.75	8, 16, 24, 32, 40, 48, 56, 64, 72, 80	
	${}^{198}_{66}\overline{C}^{3}_{D}$	3	6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66	
	${}^{200}_{50}\overline{C}^{4}_{D}$	4	5, 10, 15, 20, 25, 30, 35, 40, 45, 50	
	${}^{198}_{44}\overline{C}^{4.5}_{D}$	4.5	4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44	
	${}^{200}_{40}\overline{C}^{5}_{D}$	5	4, 8, 12, 16, 20, 24, 28, 32, 36, 40	
	${}^{198}_{33}\overline{C}^{6}_{D}$	6	3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33	
	${}^{198}_{22}\overline{C}^{9}_{D}$	9	2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22	
	${}^{200}_{20}\overline{C}^{10}_{D}$	10	2, 4, 6, 8, 10, 12, 14, 16, 18, 20	
	${}^{220}_{20}\overline{C}^{11}_{D}$	11	2, 4, 6, 8, 10, 12, 14, 16, 18, 20	

To simplify and shorten the notation of *C* instances for $C_{\mathcal{U}_{[1,100]}}$ we will use the notation C_1, for $C_{\mathcal{U}_{[50,100]}}$ we will use C_2, and so on.

With respect to standardization, *C* instances satisfy the majority of criteria. Optimal solutions and the state-of-the-art solver results can be retrieved from our repository.

5.4. B Instances

B instances are the well-recognized group of instances used for testing MWNP solvers. Originally proposed in 1998 [9], the *B* instances were defined over a configuration of three machines ($m = 3$), up to 100 jobs ($n \leq 100$), and with job processing times (precision) $p_i \in [1, 10^5]$. They are characterized by a small machine count m, and large, uniformly distributed processing times. Over time, these instances were extended to include variations in the number of machines, job counts, and processing time bounds. Therefore, *B* instances form the most extensive dataset with m extending up to 12, n taking values up to 60, and processing times up to $2^{48} - 1$ [9,40,42,115–121]. To ensure the inclusion of all notable *B*-like instances from the literature, our proposed standardized dataset includes all distributions $\mathcal{U}_{[1,2^{class}-1]}$ for $class \in \{1, \ldots, 48\}$, and all coefficients $\bigcup_{n=m+2}^{60} \frac{n}{m}$ for $m \in \{2, \ldots, 12\}$, following the formulation in [205].

Considering the generation of 100 instances per rule, as adopted in prior studies, this results in a comprehensive dataset comprising approximately 2,745,600 instances. This large volume presents significant challenges. The first challenge is the computational demand—the total time required for testing this scale of instances is significant, especially for exact solvers. The second challenge is lightness—many instances may be relatively easy for most solvers, diluting the focus on challenging cases. The third challenge is the growth in dataset size—the number of instances escalates rapidly as the maximal values for n, m, or *class* increase. The challenges are likely to remain relevant despite constant improvements in the solver performance and computational power.

Our objective is to reduce this set to include only challenging instances, thereby addressing all three problems. Previous research in OR community has indicated that instances tend to become more difficult when $\frac{n}{m} \in [2, 3]$ [95]. In the context of the MWNP formulation, AI researchers have similarly noted the importance of selecting hard instances. For instance, in [9], the authors experimentally demonstrated that for $m \in \{2, 3\}$ and job duration bounded by fixed precision ($\overline{p}$), increasing n initially raises the complexity until a peak, after which complexity begins to decrease. This trend aligns with the *phase transition*

phenomenon in $\mathcal{NP}$-hard problems. It highlights a "critical state" where approximately half of the instances are solvable to perfection and half are not, marking a peak in the computational effort. An instance is solvable to perfection (is perfect) if it has a solution in which difference between each two processor loads is maximally one. In [118], the authors extended this methodology. For fixed m and $\overline{p}$, as n grows, the number of possible schedules (m^n) expands exponentially, whereas the number of possible distinct machine loads grows linearly ($\mathcal{O}(\overline{p}n)$) leading to a critical threshold where computational resources peak. Determining whether an instance is perfect requires an extensive exploration within this critical region.

While many open questions remain, this methodology provides a practical framework for identifying generation rules for "harder" instances. Following this methodology for $m \in [2,7]$ and $D \in \mathcal{U}_{[1,10^{class}-1]}$, where $class \in [1,12]$, the authors in [118] generated 100 instances for each generation rule, increasing n until at least 50 instances were identified as perfect. Due to computational constraints, results for larger values of $class$ and m are limited.

Using a third degree polynomial regression [206] trained on data provided in [118], the authors derived the following predictive model for values of n depending on $class$ and m stored in matrix $T_{class,m}$:

$$\begin{aligned} T_{class,m} = &-0.002\,class^3 + 0.014\,class^2\,m - 0.016\,class^2 - 0.032\,class\,m^2 \\ &+ 0.812\,class\,m + 2.032\,class + 0.036\,m^3 - 0.459\,m^2 + 3.388\,m - 4.207 \end{aligned} \tag{5}$$

where $R^2 > 0.999$ predicts known values. This model provides a basis for predicting unknown values, supporting our generation rules Table 12.

Table 12. B instances with 270 different generation rules.

$class$	$notation$	$\frac{n}{m}$	m	D
$\{1,\ldots,18\}$	${}^{T_{class,m}}_{m}\overline{B}_{10^{class}}$	$\frac{T_{class,m}}{m}$	$\{2,\ldots,16\}$	$\mathcal{U}_{[1,10^{class}-1]}$

Using it we have extended their results up to $class = 18$ and $m = 16$. The results are presented in Table 13, where the bold values were originally reported in [118], and the others are predicted by Equation 5.

Table 13. Extended experimentally calculated matrix $T_{class,m}$ up to $m = 16$ and $class = 18$. Original values are bolded.

$class \setminus m$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	**4**	**7**	**9**	**11**	**13**	**16**	18	22	27	33	40	48	59	72	86
2	**9**	**11**	**14**	**17**	**19**	**22**	25	29	34	40	47	56	67	79	94
3	**12**	**16**	**18**	**22**	**25**	**28**	32	36	42	48	55	64	75	87	102
4	**15**	**20**	**25**	**28**	**31**	**35**	39	44	49	56	63	72	83	95	110
5	**18**	**24**	**29**	**33**	**38**	**42**	46	52	57	64	72	81	92	104	118
6	**22**	**28**	**34**	**39**	**44**	**49**	54	59	65	72	80	90	100	113	127
7	**26**	**33**	**39**	**45**	**50**	56	61	67	74	81	89	99	110	122	137
8	**29**	**37**	**44**	**51**	**57**	63	69	75	82	90	99	108	119	132	146
9	**33**	**41**	**49**	**57**	63	70	77	84	91	99	108	118	129	142	156
10	**36**	**45**	**55**	**62**	70	77	85	92	100	108	118	128	139	152	167
11	**39**	**50**	**59**	68	77	85	93	101	109	118	127	138	150	163	177
12	**42**	**54**	64	74	83	92	101	109	118	128	138	148	160	174	188
13	45	58	69	80	90	100	109	118	128	137	148	159	171	185	200
14	48	62	74	86	97	107	117	127	137	147	158	170	182	196	211
15	51	66	79	92	104	115	125	136	147	157	169	181	194	208	223
16	54	70	84	98	110	122	134	145	156	168	180	192	205	220	236
17	56	73	89	103	117	130	142	154	166	178	190	203	217	232	248
18	59	77	94	109	124	137	150	163	176	189	202	215	229	244	261

Neither of the mentioned papers which used some *B*-like instance set, provided instances or discussed the characteristics of the random seed values used. We propose generating 100 instances per generation rule, starting with random seed 1 as the standard. In our repository, we have already placed the generated *B* instances, organized into folders by class, as well as we provided a generator for their creation. If the researchers prefer generating instances rather than using the pre-generated ones, but only need a subset, we strongly recommend the following procedure: Generate the entire set first and then select the preferred subset. This way errors related to random seed handling should be avoided.

The described method of generating instances is both effective and well-suited for practical applications is explored in more detail in [78,92,93]. Recent theoretical advances have additionally analyzed the hardness of instances in the MWNP problem formulation [91], specifically for the $m = 2$ case [94]. Practical guidance on generating robust benchmarking tests is provided in [15].

5.5. Other Instances

In this section, we provide some additional sets of instances that do not satisfy our criteria for standardization. However, they appear in $P||C_{max}$ literature and should be described for the sake of completeness.

The first group of instances for $P||C_{max}$ problem, containing only five examples, appeared in [128]. It was used to evaluate the performance of the LPT algorithm and generated to contain hard instances for that particular algorithm.

*G*78 instance, based on the BPP and proposed in 1978 [71], is characterized with $m = 10$ and $n = 100$. The job processing times fall within the interval $[10^{10}, 10^{11}]$, which made these instance particularly challenging at the time they were introduced. For the $P||C_{max}$ problem it was used in [10,178,180]. The authors of [178] used this instance to evaluate the TS algorithm and managed to improve the result obtained by SA in [10]. Additionally, in [180], this instance is used to generate a similar set of instances with $n = 2 \cdot 10^3$ and $m = 50$ for testing his variant of TS.

Perfect packing or *PP* instances were proposed in 1995 [57] as hard instances for the purpose of testing a new HE. In this set, for every instance, the optimal schedule has equal competition time on each machine. The set covers instances with $n \leq 10^4$ and $m \in \{3, 5, 10, 15\}$. These instances were also used in [14].

MK instances were proposed 2004 [113] for testing HE algorithm. They are a union of a set of instances ${}^{15}_{5}\overline{MK}_{\mathcal{U}_{[1,100]},\mathcal{U}_{[10,100]},\mathcal{U}_{[50,100]}}$ and set of instances ${}^{10^3}_{100}\overline{MK}_{\mathcal{U}_{[1,100]}}$. These instances were also used in [17,207].

BINPACK or *BP* instances were adopted to $P||C_{max}$ in 2004 [163]. They are characterized by the uniform distribution of job duration from the range $(20, 100)$. These instances were also used in [137,140–143,177,202].

TRIPLET or *TR* were originally designed for the BPP problem [208] These instances for $P||C_{max}$ have been used for the first time in [7]. The *TR* instances are generated in such a way that the optimal solution has exactly three jobs per machine, while the job processing times belong to the $(25, 50)$ interval. They were later used in [140,163,164].

Iogra (*IO*), *It* (*IT*), and *Rand* (*RN*) instances, adapted from Multiprocessor Scheduling Problem with Communication Delays [209] and from [210] for $P||C_{max}$. Although these instances were considered easy, they are used in [166,177,188,202,211].

All the sets of instances in this section are either reused from other problems, small, or too easy in comparison with other described instances. In addition, we do not consider them as a special group because they are less explained, less used, or simply do not provide any novelty compared with *B*, *C*, *E*, and *F* instances.

5.6. Summary of Instance Groups

The key results of our SLR are the identification of the main differences between various sets of problem instances and the definition of criteria for producing standardized groups of instances for testing $P||C_{max}$ optimization algorithms. Figure 7 illustrates the usage of various types of instances over the years. As none of the "Other" group instances satisfies all standardization criteria and they are used ad hoc only, we do not present them here. The figure shows that E instances were the first in use, other types of instances were used from the 1990s. However, all types are still in use.

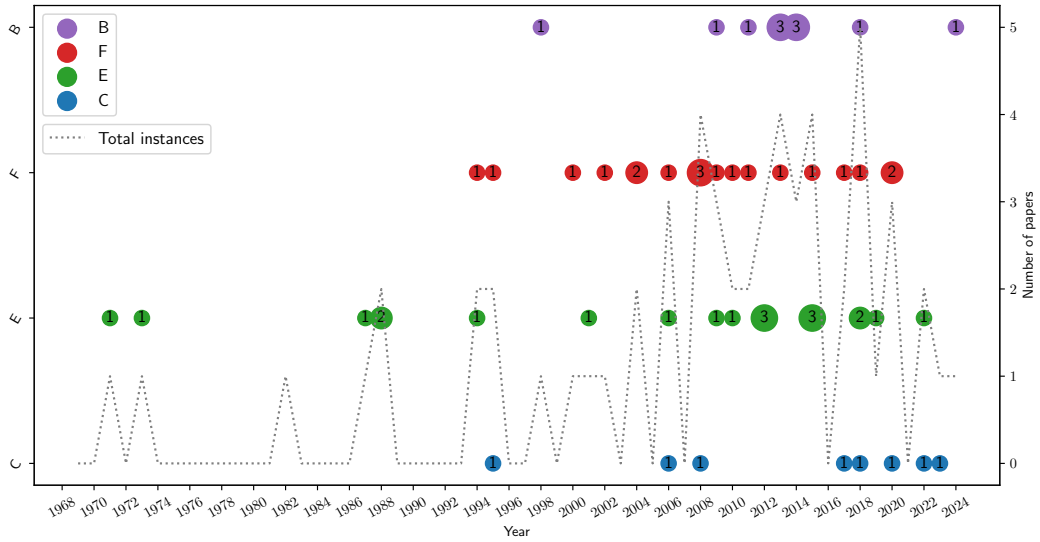

Figure 7. Used instance sets per year.

6. RQ3: Comparisons

Comparing two or more algorithms, also known as *optimization benchmarking*, is conducted in the literature with the goal of determining which algorithm has performed best on a given representative set of problem instances [212,213]. In some cases, approaches that rely only on formal analysis of the algorithms have been applied assuming that analysis will explain the quality of algorithms on all problem instances. However, for these approaches, one must presume only problem instances with a suitable structure which might not be representative of all problem instances. This is one of the shortcomings of applying formal analysis exclusively [214]. Additionally, the worst-case or average-case analysis of the algorithm's complexity does not provide complete information about how a given heuristic will perform on any given set of instances. Finally, the stochastic analysis of the algorithm's performance might require the knowledge of underlying instance distribution. This distribution, in practice might not be easily identifiable. These are the main motivations why empirical (experimental) work is necessary in order to ensure adequate comparison [214,215].

In the remainder of this section, the general methodology for comparing optimization algorithms is explained. It is then followed up with the specific details of comparing algorithms applied on $P||C_{max}$ problem. For each step of the methodology, its importance and relevance are explained, and studies in which the methodology is applied are cited. Finally, the comparisons made in the literature are presented.

6.1. General Methodology

The choice of appropriate performance metrics, along with adequately selecting parameter values and working with proper instances, form the bedrock for a robust performance

analysis of the developed algorithm. In more detail, the experimental procedure for optimization benchmarking consists of the following basic steps [212,213,216]:

1. Select performance metrics;
2. Design the experiment;
3. Select test instances;
4. Perform an experiment;
5. Analyze and present results.

Performance metrics are categorized into two essential groups: those impacting effectiveness and those influencing efficiency [217]. Effectiveness implies collecting data on solutions' quality within a limited amount of time or other computation resources (e.g., maximal number of functions evaluations). Here, the algorithm's performance is described via statistics, such as a measure of central tendencies and variability. Known measures of effectiveness are the arithmetic mean and the corresponding standard deviation or median and range. On the other hand, the efficiency of an algorithm is an estimate of runtime (or computational steps) to reach a solution of predetermined quality.

It is generally assumed that the proposed algorithm should be compared against the state-of-the-art methods for the considered problem. However, note that generating a state-of-the-art method is an iterative process, and identifying the current state-of-the-art methods could be performed only after benchmarking. Usually, the development of new algorithms is a product of fusion between old and new ideas that positively affect runtime and solution's quality. As it is hard to determine what the state-of-the-art method is, the comparison is usually conducted against some other well-known algorithms. In the absence of such methods, comparisons are conducted against the more general methods (greedy approach, integer programming). Sometimes, the comparisons are made even against a simple random restart procedure, as that may offer some reference points [216].

As mentioned above, optimization benchmarking is conducted in the literature with the goal to determine which algorithm has performed best on a given representative set of problem instances. Representative sets assume the exhaustive coverage of different cases for any important characteristic of the instances.

Evaluating algorithm performance can be performed through graphical analysis or statistical testing. During the comparison process, the authors should always aim to achieve fairness. As there are various aspects of the study that may influence the fair comparison, we list a few recommendations provided in [213]:

1. Avoid comparing tuned versus untuned algorithms. Namely, the best scenario is if algorithms are compared at their peak performance, i.e., after parameter values are determined via some tuning procedure.
2. Whenever possible, conduct a comparison on the same machine, with respect to both hardware and software characteristics. If this is not possible, use a machine with similar characteristics and perform the appropriate scaling of the results.
3. The analysis must adhere to the 3Rs of Data Science: repeatability, reproducibility, and replicability all pointing towards getting the same results. Repeatability involves the same researcher and environment. Reproducibility engages another researcher on the same computer system. Replicability means that the estimated performance could be achievable by anyone in any computing environment.

6.2. $P||C_{max}$ Problem Specific Methodology

The most important part of any computational evaluation of an algorithm's performance is to address the correct measure of performance. A very important moment for practical comparison of any proposed algorithm is the quality of its implementation. Additional information related to MH algorithms can be found in [218]. The usual performance

metrics for $P||C_{max}$ problem-specific comparisons are the quality of the solution and the running time, where the time metric could refer to the following:

- time to complete the execution (until the stopping criterion is fulfilled) in the majority of the reviewed literature and explained in [123];
- time required to find the best solution (time-to-best) in several reviewed papers and explained in [12].

Sometimes the time of execution of some parts of the optimization algorithm can be relevant, for example, the time can be measured for lower bound values [95], initial solution construction [122], etc.

Measurement of the solution quality may be more difficult, especially in cases when the optimal solution of the given instance is not known. The quality of solution metrics used in the literature for $P||C_{max}$ are as follows:

- *makespan*—the objective function value [6,110,111,148,188]. This is the simplest solution quality measure, used mostly in the cases when the optimum or some other relevant data are not known. The makespan values obtained by analyzed algorithms are compared and the lowest among them is declared as the best. The main disadvantage of this metric is that it does not give any information about the distance of compared solutions from the best possible (optimum) solution.
- gap (error)—relative distance between the compared solutions [5,8,14,20,21,26,95,113,114,133–139,141,142,145,148,153,157,159,163,164,174,177,182–184,189,190,193–200,204,219,220]. The gap is calculated as $\frac{makespan-bound}{bound}$, where bound is usually some estimation for the makespan of the best possible solution (LB); however, it can also be the makespan of the solution provided by some other algorithm (UB). Other than having the same drawback as the previous metric, the main disadvantage of using the gap is that it can be calculated in different ways. When calculated differently, there is no consistent way of establishing a fair comparison.
- opt—achieved optimality flag [5,12,20,95,114,139,141,142,144,145,148,161,163,164,166,183,194,220]. When the optimal makespan value is known, this flag indicates whether the considered algorithm(s) managed to provide the optimal solutions or not.
- OPT—proved optimality [12,57,63,113,118,123,124,137,139,144,164,207]. This metric differs from the previous one in the fact that the evaluated algorithm is able to prove the optimality of the provided solution.

There are some additional algorithm performance measures used in the relevant literature. For example, some exact solvers are characterized by the number of solved LP relaxations or the number of branches (nodes) required to find optimal solutions and prove their optimality [8,14,17,63,110,111,113]. Iterative algorithms perform better when they require small number of iterations to provide high-quality solutions [6,12,166,178,180,202]. Stochastic algorithms' performance should be evaluated also by means of statistical tools illustrating their stability with respect to the value of *seed* used for initialization of random number generator. This requires a sufficient number of executions (runs, restarts) to have the corresponding values of mean, standard deviation, and other statistical data meaningful for the derived conclusions [213,217].

Other than the statistical data related to the repeated execution of (stochastic) the algorithm on a single instance, the majority of papers usual present summary (average) performance measures for a set of instances from the same class e.g., [123,207]. These kinds of measurements can testify to the robustness of the analyzed algorithm with respect to the characteristics of different instances.

For parallel algorithms, it is important to take into account the number of processing elements involved in the algorithm execution [202,221,222]. To ensure fairness in comparison,

the stopping criterion is usually reduced appropriately. However, sometimes, the goal is to examine if the enriched hardware resources can contribute toward the improvement in the final solution quality. In addition, it could be very informative to examine the contribution of each part of the algorithm (executed in parallel) and to modify them if necessary.

In order to conduct a comparison between two or more optimization algorithms for $P||C_{max}$ the next step involves the selection of benchmark instances in such a way as to allow some sort of generalization. As the answer to Research Question 2 (Standardization of the problem instances), discussed in Section 5, we recommend using any of the four instance types B, C, E, and F, preferably all four of them. Finally, in the remainder of the section, we provide comparisons of specific algorithms for $P||C_{max}$ problem.

6.3. Comparisons of Specific Algorithms $P||C_{max}$ Problem

In this section, we provide information about the existing comparisons between various algorithms. Usually, algorithms from the same group are compared, although sometimes we also found the comparison between algorithms belonging to different groups. The remainder of this section follows the categorization and grouping from Section 4, with these non-typical cases pointed out when appropriate. For each group of methods, the types of comparisons are presented first. Then, we try to establish a relationship graph among the algorithms based on the available comparisons. The nodes in that graph represent the algorithms, while the direct edges source represents the "better" algorithm in that comparison.

The experimental results on the same group of standardized instances are used to evaluate the performance of one algorithm relative to others. The performance measures used to build a graph are *gap* and the number of instances for which the optimum is reached/optimality proven. Having in mind the drawbacks of *gap* calculations, we have also included the elapsed time and the number of iterations, if needed and provided in the literature. When elapsed time is taken into account, it was normalized based on known characteristics of the resources. In extreme cases, when it is not possible to conclude which algorithm among those examined is better, we did not connect them by the directed edge. Transitive relationships were not shown in the graphs. Therefore, the position in the graph (the higher the better) is only an estimate of the quality of the algorithm.

6.3.1. Comparisons of Exact Algorithms

In the reviewed papers, *Exponential exact algorithms* have pure theoretical importance and usually are not implemented at all. In addition, due to their complexity, it is not rewarding to compare their performance on (benchmark) problem instances. For their theoretical comparison, standard metrics (WTC and WSC) are utilized. However, there are some exceptions.

The DP_{87} algorithm was compared with the HE algorithms BIN and DM in [57]. It was significantly slower than the other two algorithms for instances ${}^{50}_{3}\widetilde{C}_{1-5}$. It is also compared with the LPT and RAS_2 CHs, as well as RAS_1 IH on instances ${}^{50}_{5}I_{\mathcal{U}_{[10,100]}}$ in [136]. As expected, the other algorithms demonstrated superiority with respect to solving times. The second practical problem with this algorithm mentioned in [136], relates to its memory requirements for instances with $m = 5$ and $n \geq 30$. More precisely, the machine used for experimental evaluation did not have enough memory to complete the experiment.

The practical importance of *Fixed parameter tractable algorithms* is also very small, and therefore, they are compared mainly using the theoretical metrics WTC and WSC. However, theoretical metrics cannot estimate the performance of any particular algorithm on any tested instance. Despite bad theoretical indicators, an algorithm may provide an optimal solution very quickly if the instance turns out to be "easy" for that algorithm. On the other

hand, having experimental results for different sets of instances enables the application of machine learning techniques to predict the difficulty of any new instance that has not been tested yet as it is conducted in [1].

Hybrid exact algorithms are, contrary to the first two groups, focused on solving some particular instances. Performance evaluation of these algorithms is conducted by numerical experiments involving selected sets of benchmark instances. Therefore, general performance could not be judged outside the tested set of instances. In addition, the published results cannot be repeated or reproduced easily, because the algorithms' implementations, as well as the utilized instances, are not publicly available. Consequently, there are not many papers that report on comparing these algorithms, especially those that involve more than two HE algorithms.

BIN and *DM* algorithms, presented in paper [57], have been compared to each other and with DP_{87} EE. The first comparison is conducted on a subset of an early version of *C* instances, ${}^{50}_{3}\widetilde{C}_{1-5}$. As the best-performing algorithm, DM was identified, while BIN sometimes was an order of magnitude slower. DP_{87} performed the worst and it was practically unusable for instances with $m \geq 3$. In the second experiment, the authors used ${}^{10^4}_{15}\widetilde{C}_{1-5}$ instances. Although not being able to solve all instances, DM clearly outperformed BIN, except for some very small instances. In the third experiment, the authors used *PP* instances and made the same conclusion.

CGA and *CKK* [9] algorithms were compared with each other on instances ${}^{200}_{2}\widetilde{B}_{10^{10}}$ and ${}^{100}_{3}\widetilde{B}_{10^5}$ where always $m = 3$. In both cases, CKK outperforms CGA. Especially on the *perfect instances*, the difference in performance is significant.

CP_{04} algorithm [113] is compared with simple ILP solver. For the experimental evaluation, memory was limited to 50 Mb, while the running maximum time was set to $2 \cdot 10^3$ s for both algorithms. Experiments were performed on *MK* instances. The first experiment involved ${}^{15}_{5}\overline{MK}_{\mathcal{U}_{[1,100]},\mathcal{U}_{[10,100]},\mathcal{U}_{[50,100]}}$ instances. The obtained results show that both algorithms were always able to converge to the optimal solution, with CP_{04} requiring less computational time for the largest instances. On the other hand, the ILP solver was faster for the smallest instances. The second experiment was performed on the ${}^{10^3}_{100}\overline{MK}_{\mathcal{U}_{[1,100]}}$ instances. In this case, the ILP solver was not able to find optimal solutions in the majority of cases, while CP_{04} provided optimal solutions for almost all instances with drastically smaller computation effort. The authors of [207] critically reviewed CP_{04} and showed that the algorithm can be significantly outperformed by DM on the same sets of instances.

The HJ algorithm [14] is compared with DM. As the starting bounds, MSK CH and $\vec{L}_{FS}$ were used. The *PP* instances were used in the first experiment. Due to the very good starting bounds, HJ solved to optimality in all of the instances in a very short time, and branching was required only for three instances. On the other hand, DM failed to solve 13 instances. The second experiment involved ${}^{10^4}_{15}\widetilde{C}_{1-5}$ instances from [57]. Both algorithms exhibited similar behavior; however, five instances remained unsolved by HJ, which is again better comparing to eight instances unsolved by DM. In the third experiment, HJ showed very good performance on *F* instances, although not being directly compared with any other algorithm. The authors concluded that it solved to optimality 13 instances being open problems for some time. The algorithm was not able to solve 3 not-uniform and 18 uniform instances. In the fourth experiment, HJ algorithm was tested on ${}^{200}_{80}\widetilde{C}^{2.5}_{\mathcal{U}_{[\frac{n}{5},\frac{n}{2}]}}$ instances. The authors were not satisfied with 35% of unsolved instances in this case and concluded that an alternative approach should be developed.

DIMM, proposed in [114], is compared with DM on *F* instances. The main ingredients of DIMM include $DIMM_{SS}$ MH and MT1 [151] Knapsack solver. The obtained results

show that DIMM solved all instances to optimality within small computation times, clearly outperforming DM.

SNP_{ie} and *RNP* [115] algorithms were compared with each other and with CKK and CGA algorithms. In the first experiment, the instances ${}^{40}_{3}\widetilde{B}_{10^7}$, with $m = 3$ and $n \geq 25$, were used. According to the provided results, SNP_{ie} significantly outperforms CKK with respect to running time. Authors mentioned (without providing profs) that for $m = 4$ and $m = 5$ CGA is faster than CKK, and in the second experiment, they compared SNP_{ie} and RNP only with CGA on instances ${}^{33}_{4}\widetilde{B}_{10^5}$, with $m = 4$ and $n \geq 20$. CGA is outperformed by several orders of magnitude by the other two algorithms. RNP showed better performance than SNP_{ie}. The third experiment was performed for the same algorithms on instances ${}^{30}_{5}\widetilde{B}_{10^4}$, with $m = 5$ and $n \geq 20$. The obtained results are similar to those in the previous experiment. Finally, an additional experiment is carried out involving instances ${}^{40}_{5}\widetilde{B}_{10^8}$, with $m > 2$ and $n \geq 20$. Again RNP was the best algorithm, followed by SNP_{ie}.

IRNP [116] was compared with RNP on instance sets ${}^{52}_{6}\widetilde{B}_{2^{31}}$ and ${}^{40}_{10}\widetilde{B}_{2^{31}}$. On all instances, IRNP was a few orders of magnitude faster than RNP.

MOF [117] was compared with IRNP and demonstrated its superiority on ${}^{40}_{10}\widetilde{B}_{2^{31}}$ instances.

BSBCP, CKK_i, RNP_i, IRNP, and CGA algorithms are compared in [118]. For $m = 2$, the authors showed that CKK dominates CGA. Without providing experimental results, the authors mentioned that BSBCP performed better than IRNP for $m > 7$. Finally, on instances ${}^{50}_{7}\widetilde{B}_{2^{48}}$, with $m > 2$ and $n \geq 25$, RNP_i clearly outperformed IRNP.

BSIBC [119] was compared with BSBCP and DIMM. The first experiment was conducted for BSIBS and BSBCP algorithms on ${}^{45}_{20}\widetilde{B}_{10^{15}}$ instances. Both algorithms performed well, with BSIBC solving the majority of the instances faster. In the second experiment, the authors compared different variants of the BSIBC algorithm against DIMM on a subset of F_U recognized to contain hard instances in [114]. The conclusion was that BSIBC with limited discrepancy search is able to outperform DIMM by up to three orders of magnitude.

SNP_{ess} and *HI14* [120] were compared with each other and with MOF on ${}^{50}_{10}\widetilde{B}_{2^{48}}$ instances. SNP_{ess} is always faster than MOF, except for instances with small n. As expected, the authors reported smaller computation times for HI14 in comparison to other algorithms for all instances.

CIW [121] was compared with SNP_{ess}, MOF, and BSBCP, and clearly outperformed all other algorithms: SNP_{ess} on ${}^{60}_{7}\widetilde{B}_{2^{48}}$ instances, MOF on ${}^{60}_{10}\widetilde{B}_{2^{48}}$ instances, and BSBCP on ${}^{60}_{12}\widetilde{B}_{2^{48}}$ instances.

WL [8] was compared with DIMM and HJ solvers. First, the comparison was conducted on ${}^{50}_{15}\widetilde{C}_{1-5}$ instances. According to the authors, this subset of the first version of *C* instances [57] contains difficult test examples. In this experiment, WL was compared with HJ and was faster on all instances except the biggest ones. However, in total, HJ solved more instances within less running time. In the second experiment, HJ outperformed WL on *F* instances, although both algorithms had plenty of unsolved instances. In the third experiment, WL was compared with DIMM on *F* instances and it was shown that WL can solve three instances faster than DIMM. However, DIMM was generally a superior approach. The fourth experiment involved comparison of WL with HJ on ${}^{200}_{80}\widetilde{C}^{2.5}_{\mathcal{U}_{[\frac{n}{5},\frac{n}{2}]}}$ instances presented in [14]. It revealed that WL has comparable performances with HJ. The fifth experiment shows clear superiority of the WL algorithm in comparison with its variant WL′ on an extended set of ${}^{30}_{15}\widetilde{C}_{1-5}$ instances from the first experiment. An additional comparison of WL and WL′ on a more expanded ${}^{100}_{25}\widetilde{C}^{2,2.5,3,4,5}_{1-5}$ instances was performed in [63]. The authors made the same conclusion: WL outperforms WL′.

The *LCS* algorithm [42] in comprehensive practical experiments was compared with CGA, RNP, IRNP, SNP_{ie}, BSIBC, BSBCP, SNP_{ess}, MOF, and CIW on $\widetilde{B}$ instances with $n \geq 20$. The obtained results can be summarized as follows: SNP_{ess} is dominant for $m = 3$ and for small values of n when $m \in \{4, 5\}$. CIW dominates for large values of n when $m \in [4, 12]$. SNP_{ie} performs the best for $m \in [8, 10]$ if the values of n are very small. BSIBC and BSBCP outperform others for $m \in \{11, 12\}$ and small values of n. For $m \in [6, 12]$ and $n \leq 35$, MOF often performs the best. From the presented results, it is clear that CIW dominates other algorithms in the majority of cases. LCS performs better on instances with $m > 5$ and requires less memory for $m > 8$.

KL [122] was compared with DM and DIMM on F instances. As expected, DM was outperformed by DIMM which managed to solve all instances in a much shorter time. KL also solved all instances in significantly less time than DIMM. In addition, it did not require branching: the initial heuristic KL_h was enough.

AF [123] was compared with HJ and WL solvers. Comparison with HJ was performed on instances similar to a subset of C instances with $\frac{n}{m} = 2.5$ presented in [14]. AF dominated HJ by solving all instances to optimality. Comparison with WL involved the whole set of C instances. In this case, AF outperformed WL, it successfully solved 77 out of 3500 instances.

iAF [124] was compared with the AF solver on C instances and clearly improved AF results by solving all but one instance within a shorter execution time.

DIST [12] was compared with iAF solver on C instances with $\frac{n}{m} = 2$. Within the subproblem time limit of $0.06s$ and 30 runs, DIST optimally solved all instances in a much shorter execution time. It is important to note that in some cases DIST was not able to guarantee the optimality of the provided solutions.

Based on the described empirical results given in papers included in our SLR, the relationship between all exact algorithms is illustrated in Figure 8. Blue color represents HE algorithms, and light blue represents EE algorithms. The first observation is related to a separation of the graph into two parts (left and right), connected just with the $\overline{F}_U$ branch. The right part of the graph is characterized by algorithms oriented to solving instances with small m and large job processing times. They were compared just on sets of $\widetilde{B}$ instances. The left part of the graph is focused on algorithms for solving instances with larger m, and smaller job processing times. On the left side, a bigger variety of instances can be seen, primarily F and variants of C. These two types of algorithms had been developed separately, the first for the purpose of solving $P||C_{max}$ and the second for solving MWNP. Only one comparison between these two types of solvers is performed, between BSIBC and DIMM algorithms on $\overline{F}_U$ instances, with BSIBC being superior. Due to the lack of detailed comparisons in the literature, we did not have enough results to make clear relations between algorithms. Therefore, we made a compromise by keeping several algorithms on the same level of the comparison graph in Figure 8. This is specially related to the right part of the graph.

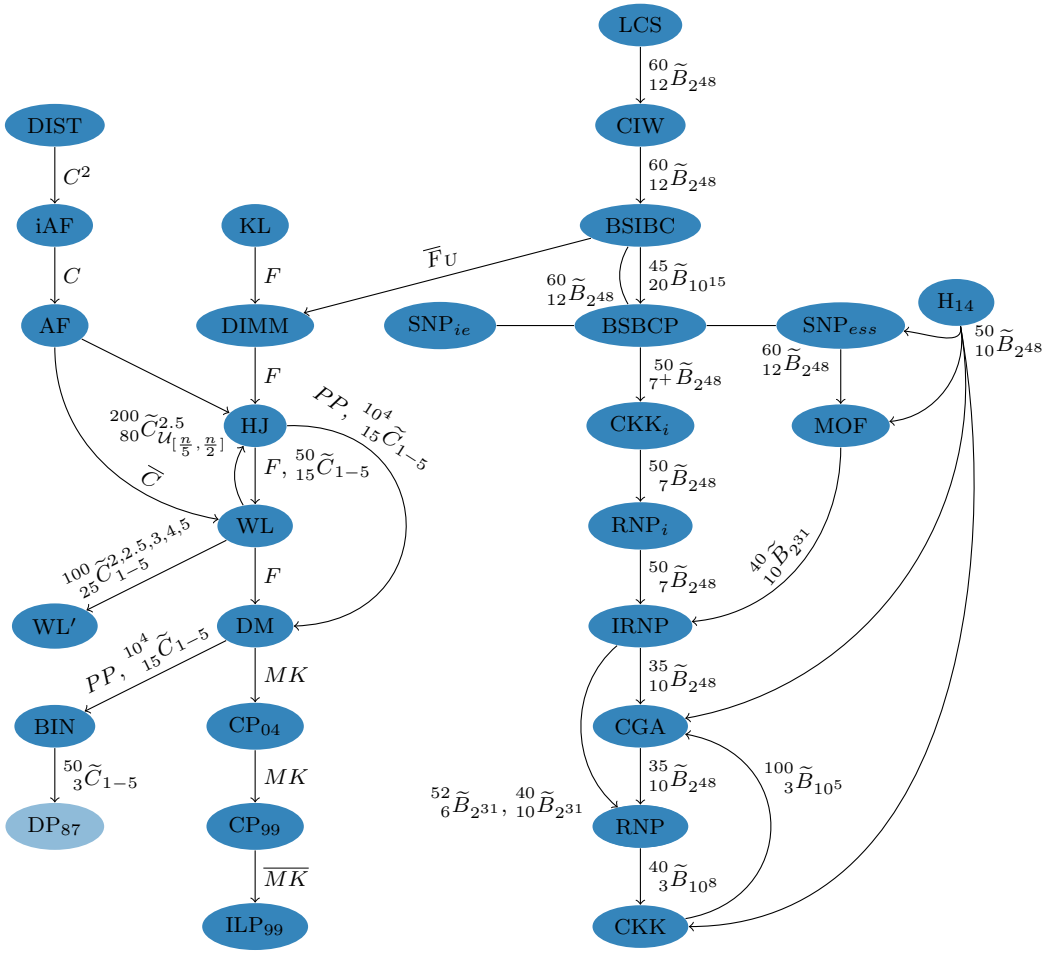

Figure 8. Graph of experimental comparisons of exact algorithms.

Final remark about the experimental evaluation of exact algorithms, is the importance to utilize standard benchmark instances and report various metrics for quality of solution under the same conditions. This will ensure fair and comprehensive comparison.

6.3.2. Comparisons of Heuristic Algorithms

The comparison of *Constructive heuristics* is usually very easy as there are many proposed comparison metrics. Both theoretical and experimental comparisons are relevant. For theoretical comparison, WTC and AR metrics are utilized, while practical comparison involves similar metrics as for HE algorithms.

If $n > m$ is assumed, WTC can be expressed as dependent only on n, which enables sorting of the CH algorithms in the non-decreasing order as follows. P_1: $\mathcal{O}(n)$; SLS, LPT, MAAAT, RAS_2, AP, LPT_R, PSC, PSC_i, SPS, and SLACK: $\mathcal{O}(n \log n)$; MF′, MF, MF_e, MF_i, and COMBINE: $\mathcal{O}(kn \log n)$; MPS: $\mathcal{O}(n^2)$; LDM: $\mathcal{O}(n^2 \log n)$; FGH, DGH and PSMF: $\mathcal{O}(kn^2)$; LISTFIT and DJMS: $\mathcal{O}(kn^2 \log n)$; MS: $\mathcal{O}(n^3)$; SS: $\mathcal{O}(n^2 S)$; H_1: $\mathcal{O}(n^2 \log(\frac{1}{\varepsilon} + \frac{1}{\varepsilon^2}))$; and finally, KK has $\mathcal{NP}$ WTC. The meaning of parameters k, S, and ε is explained in Table 4.

Comparing CH algorithms with respect to AR is not always straightforward. The AR of LPT equals $\frac{4}{3} - \frac{1}{3m}$, which is presented when the algorithm was introduced [128] and again proved in [223]. However, for some of the CH algorithms, AR is not easy to calculate. The AR for MF has been improved from $1.22 + 2^{-k}$ [129], to $1.2 + 2^{-k}$ and $\frac{13}{11} + 2^{-k}$ for $m \geq 13$ [224], and finally to $\frac{13}{11} + 2^{-k}$ for arbitrary m [225]. AR for LDM was an open problem for a long time, and first result was given for $m = 2$ [226], while the most recent analysis of general case was provided in [227], with the conclusion that AR should belong to the interval $\left[\frac{4}{3} - \frac{1}{3(m-1)}, \frac{4}{3} - \frac{1}{3m}\right]$. Additional results can be found in [228–231].

If it is assumed that m and k are approaching infinity, AR can be expressed as constant, and the CH algorithms can be sorted in the non-increasing order of AR values as follows. SLS, KK, LPT_R: 2; Specific case for MPS and SPS: $1 + \frac{1}{z}$ where $z > 1$, so upper bound of $\frac{3}{2}$ can be used. LPT, LDM, RAS_2 and SLACK: $\frac{4}{3}$; MF': $\frac{5}{4}$; MF, COMBINE, LISTFIT, FGH, DGH, DJMS and PSMF: $\frac{13}{11}$; MF_i: $\frac{72}{61}$; Specific values are provided for $m = 2$ for some CHs: LPT, LDM: $\frac{7}{6}$; SLACK: $\frac{9}{8}$ and finally MF_e and COMBINE: $\frac{10}{9}$.

There are also some additional asymptotic performance indicators for LPT [232–237], and for MPS [238]. The statistical asymptotic performance is analyzed for SLACK [239], for LPT [86,240–246], for some other LS variants [243,246–249], and in general [250]. However, the obtained results involve some parameters that are not always available. Therefore, these results are not utilized in this SLR.

Theoretical performance indicators illustrate the expected solution quality and required running time of each particular CH. However, the practical behavior of these algorithms may differ significantly. For experimental evaluation, achieved solution quality (the makespan value) and the running time spent for each problem instance are usually the most relevant performance indicators. The majority of the reviewed papers report on experimental evaluation. However, usually, only a few CHs are compared simultaneously, on a very specific set of problem instances, which makes the conclusions non-adequate in many situations. Some of the conclusions drawn by the authors of reviewed papers and related to the experimental comparison of CH algorithms are presented here.

The variant of *MF* algorithm [129] where the parameter k takes value 7 (MF_7) was compared with LPT and MF'_7. The quality measure was defined as the average gap with respect to the lower bound. The comparison was made on a small instance set ${}^{30}_{10}I$. Based on the presented results, the authors concluded that MF_7 was clearly the best and MF'_7 performed better than LPT in the majority of cases.

The MF_e algorithm [133] was compared with MF, LPT; and the ε-DUAL PTAS algorithm with $\varepsilon = \frac{1}{5}$. The first version of E_2 instances with mean 450, where variance is not explicitly provided, but limits for values are in interval $[100, 800]$ (denoted as $\widetilde{E_{2,\mathcal{N}_{450}}}$) was used. The experimental results showed that MF_e performed generally the best, MF outperformed the other two algorithms, and LPT was better than ε-DUAL.

The *COMBINE* algorithm [134] was compared with MF and LPT algorithms, on the same variant of E_2 instances as in [133]. It was shown that COMBINE provides better results on average, which is expected considering that it represents a combination of the other two mentioned algorithms. In addition, it was confirmed that MF outperforms LPT.

The performance of the RAS_2 algorithm [136] was compared with LPT and MF; RAS_1 IH; and DP_{87} EE solver. The first experiment is conducted on instances ${}^{50}_{5}I_{\mathcal{U}_{[10,100]}}$. In this experiment, it is shown that RAS_2 performed better than LPT, slightly worse than RAS_1. In addition, its running times are significantly shorter compared to DP_{87}, where the exact solver failed on some instances, due to the memory limitations of the used machine. In the second experiment, tests are conducted on instances ${}^{2\cdot 10^3}_{150}I_{\mathcal{U}_{[10,100]}}$. In comparison with LPT, RAS_2 obtained better quality solutions within slightly longer computation times. With respect to RAS_1, RAS_2 provided marginally worse solutions within significantly shorter computational times. Based on the results of the third experiment on instances ${}^{250}_{50}I_{\mathcal{U}_{[10,100]}}$ and the fourth experiment on instances ${}^{10^4}_{250}I_{\mathcal{U}_{[10,100]}}$, it can be seen that RAS_2 is similar to MF in terms of the solution quality. However, RAS_2 provides a significant reduction in CPU times.

The *LISTFIT* algorithm [137] shows considerably better average performance compared to LPT, MF, and COMBINE on the E instance set. As expected, COMBINE outperforms MF, and MF outperforms LPT.

Among all variants of *FGH* and *DGH* algorithms [130], npar-FGH$_{10}$, npar-DGH$_1$ and npar-DGH$_2$ were evaluated. The listed algorithms have been compared with LPT, MF (MF$_7$ and MF$_{16}$), RAS$_2$; and RAS$_1$ IH on their own instance sets, with sizes up to ${}^{10^4}_{10^3}I_{\mathcal{U}_{[1,10^3]}}$. The authors utilized the gap with respect to L_2 as a performance measure. As can be seen from the presented experimental results, the listed algorithms dominate for the majority of instances. However, in some cases, MF algorithms or RAS$_1$ performed the best. There was no significant difference in the computational time between the compared heuristics. Upon the proper parameter tuning, no obvious difference between the FGH and DGH algorithms can be found. Although they did not usually produce identical solutions, the computed gap deviations were similar.

AP_{10}, AP_{ni}, and AP_i (AP) algorithms [138] were compared with LPT, FGH, DGH, and MF. Instances set ${}^{10^3}_{50}I_{\mathcal{U}_{[1,100]}}$ were used in the first experiment, while the second experiment was conducted on instances ${}^{10^3}_{50}I_{\mathcal{U}_{[1,10^3]}}$. The authors concluded that the algorithms they proposed require much less time to run than FGH, DGH, and MF. According to the solution quality, AP algorithms showed superiority only in comparison with LPT.

H_1 [139] has not been directly compared with other algorithms. Instead, the authors mentioned that it was capable of finding 556 out of 780 optimal solutions on the *F* instances when using lower bounds L_3, L_θ, and L_{HS}.

The *SS* algorithm [95] was compared with MS, the better among MF and LPT; and MSS IH with respect to the solution quality. Instance sets ${}^{200}_{80}\widetilde{C}^{2.5}_{\mathcal{U}_{[\frac{n}{5},\frac{n}{2}]}}$ and ${}^{100}_{15}I_{\mathcal{U}_{[50,100]}}$ were utilized. MSS performed the best, followed by SS and MS. In the case when SS was used (instead of LPT_R) to construct an initial schedule for the MSS IH, even better solutions were achieved; however, at the cost of increased computation time.

The *MPS* algorithm [140] was compared with LPT. It produced better AR on *BP*, *TR*, and F_{NU} instances than LPT. However, LPT showed slightly better results in terms of execution time. The issue with these experiments is that the authors do not provide insight into makespan values or other directly dependent statistics, making it difficult to compare MPS with other algorithms based on solution quality.

Based on the solution quality, the *PSC* algorithm [141] was compared with LPT; and the 3-PHASE and 1-SPT IHs. Comparison with LPT and 3-PHASE was conducted on *F* instances. On the other hand *BP* instances were utilized for comparison with LPT and 1-SPT. The provided experimental results show that PSC outperforms LPT on both instance sets. With respect to both IHs, PSC exhibits comparable performance.

PSC_i (the best performing among all presented variants) [142] was compared with LPT; and 3-PHASE and K-SPT IHs under the same conditions as in the previous paper. Regardless of the performed modifications, the conclusion of the experimental evaluation remains unchanged.

A similar experimental evaluation was performed for the *SPS* algorithm [143], and a similar conclusion was made. SPS performed better than LPT on *BP* and *F* instances. It was comparable with 3-PHASE on *F* instances and with 1-SPT on *BP* instances.

The *DJMS* algorithm [144] was compared with LPT, MF, COMBINE, and LISTFIT on *E* instances. It showed the smallest overall average percentage error compared to LPT, MF, COMBINE, and LISTFIT, although LISTFIT performed slightly better with respect to the percentage of instances solved to optimality.

PSMF algorithm [145] is evaluated against LPT, MF, COMBINE, and LISTFIT on *E* instances. It can be seen that PSMF performed the best, and it flowed with LISTFIT, COMBINE, MF, and LPT, respectively. In addition, authors used the F_{NU} set to compare PSMF with ME (the best result of 1-SPT, 1-BPT, K-SPT, and K-BPT), PSMF+ and HI IHs; and DM HE. In this experiment, HI always outperforms all other algorithms. PSMF obtains the worst average relative error (the same order of magnitude as ME and DM) and solves the fewest instances to optimality. PSMF+ performs better than PSMF with respect to

the average relative error and the number of provided optimal solutions with negligible increase in the running time. Actually, it can be ranked as the second the best algorithm, outperformed only by HI.

The *SLACK* algorithm [146] was compared with LPT, COMBINE, and LDM on F instances. It produced better overall experimental results than LPT and COMBINE but not better than LDM. COMBINE outperformed LPT. The SLACK+ IH was also compared with LDM, showing generally better results on F_{NU} instances but weaker results on F_U instances, with computational times more than an order of magnitude longer in both cases. The issue with these experiments is that they provide only win/equal/lose results without concrete makespan or gap values, making a more precise comparison with other algorithms impossible.

The studies in [41,219] compare LPT, MF, COMBINE, and LISTFIT on the instance set ${}^{300}_{10}I_{\mathcal{U}_{[1,100]}}$. The results indicate that LISTFIT achieves the best performance, followed by COMBINE and MF, with LPT performing the worst, which is consistent with previous findings.

In [113], the author examined the performance of the CP_{04} HE algorithm exploring various CHs for generating the initial solution: LPT, FGH_1, MF_{17}, and AP_{10}. Instance set ${}^{10^3}_{15}I_{\mathcal{U}_{[1,100]}}$ is used for the evaluation. Based on the experimental evaluation, it can be concluded that CP_{04} with AP_{10} produced the best results on hard instances, while better results can be obtained on easy instances starting from MF_{17}, the obtained initial solution. However, the provided results cannot be used for direct comparisons of the given constructive heuristics.

WTC and AR are also standard performance measures for theoretical comparison of *Improvement Heuristics*. Their experimental evaluation involves the quality of the produced solution and the required computational time on the selected test instance. However, WTC may not be known for some IHs, as well as AR for IHs that start from a random initial solution. On the other hand, AR which originates from CH, used to obtain an initial solution for the improvement procedure, may not be improved in a theoretical sense. In those cases, experimental comparison is the only choice.

Assuming $n > m$, enables to express WTC as dependent only on n and sorts the IH algorithms in non-decreasing order with respect to complexity as follows: IC, ICI, ICII: $\mathcal{O}(n \log n)$; KOMP: $\mathcal{O}(n^2 S)$; LPT+, MF+: $\mathcal{O}(n^2 S \log n)$; EX: $\mathcal{O}(n^3 S)$; MSS: $\mathcal{O}(n^2 k S^2)$; MSK: $\mathcal{O}(n^3 k S^2)$; X-TMO and finally DIST with WTC of $\mathcal{NP}$ complexity.

For some of the IHs, WTC estimations have not been provided in the related papers. Therefore, the necessary derivations were performed to complete the above-presented list. For example, in [157] WTC of EX algorithm, was considered only for the 2-machine case and it is estimated to $\mathcal{O}(n^2)$. In [158], precise pseudo-code for the case $m > 2$ is provided, allowing us to derive WTC. Assuming that every improvement minimally decreases the load of the most loaded machine (i.e., by 1), the number of iterations in the main loop is related to the sum of job durations (S). In the worst case the improvement will always occur for the last considered pair of machines, implying that WTC cannot be worse than $\mathcal{O}(n^2 m S)$, i.e., $\mathcal{O}(n^3 S)$ for $n > m$. Similarly, the WTC for MSS, MSK, LPT+, and MF+ could be calculated. Estimation of the WTC for the 3-PHASE algorithm is more complicated, as mentioned in [135]. The same holds for HI [139], and therefore the corresponding algorithms are missing in the aforementioned list.

If it is assumed that m and k approach infinity, AR can be expressed as a constant. Thus, the IH algorithms can be sorted in the non-increasing order of AR, with their constructive heuristics in parentheses: X-TMO (MF), MF+ (MF), PSMF+ (PSMF): $\frac{13}{11}$; HI (ε-DUAL): $\frac{5}{4}$; EX, ICII, LPT+ (LPT), MMIPMH (LPT), SLACK+ (SLACK), DIST (SLACK): $\frac{4}{3}$; ICI: $\frac{3}{2}$; IC, 3-PHASE, MSS (LPT_R), MSK (LPT_R): 2.

A comparison of CHs and IHs with respect to WTC and AR can be found in Table 14. The question mark (?) in the second or the third column indicates that the information about the corresponding performance indicator is missing. The algorithms are graded with respect to WTC and AR. The corresponding points are presented in the fourth and fifth columns, respectively. A larger number of points indicates a higher grade. If the performance is missing, the grade of the corresponding algorithm is set to 0. Finally, WTC and AR points are summed up (values presented in the last column) defining the overall grade of algorithms. In Table 14, the algorithms are sorted in the non-increasing value with respect to the sum of points. Regarding the experimental evaluation of the IH algorithms, proposed SLR revealed the following information.

Table 14. Heuristic algorithms comparison by AR and WTC. "?" represents unknown values.

Name	WTC	AR	WTC Points	AR Points	Σ Points
MF_i	$\mathcal{O}(kn \log n)$	$72/61$	13	7	20
COMBINE	$\mathcal{O}(kn \log n)$	$13/11$	13	6	19
MF	$\mathcal{O}(kn \log n)$	$13/11$	13	6	19
SLACK	$\mathcal{O}(n \log n)$	$4/3$	14	4	18
MF′	$\mathcal{O}(kn \log n)$	$5/4$	13	5	18
LPT	$\mathcal{O}(n \log n)$	$4/3$	14	4	18
RAS_2	$\mathcal{O}(n \log n)$	$4/3$	14	4	18
ICII	$\mathcal{O}(kn \log n)$	$4/3$	13	4	17
SPS	$\mathcal{O}(n \log n)$	$3/2$	14	3	17
ICI	$\mathcal{O}(kn \log n)$	$3/2$	13	3	16
FGH	$\mathcal{O}(kn^2)$	$13/11$	10	6	16
DGH	$\mathcal{O}(kn^2)$	$13/11$	10	6	16
PSMF	$\mathcal{O}(kn^2)$	$13/11$	10	6	16
SLS	$\mathcal{O}(n \log n)$	2	14	2	16
P_1	$\mathcal{O}(n)$	?	15	0	15
MPS	$\mathcal{O}(n^2)$	$3/2$	12	3	15
LDM	$\mathcal{O}(n^2 \log n)$	$4/3$	11	4	15
LISTFIT	$\mathcal{O}(kn^2 \log n)$	$13/11$	9	6	15
DJMS	$\mathcal{O}(kn^2 \log n)$	$13/11$	9	6	15
MAAAT	$\mathcal{O}(n \log n)$	?	14	0	14
AP	$\mathcal{O}(n \log n)$	?	14	0	14
LPT_R	$\mathcal{O}(n \log n)$	?	14	0	14
PSC	$\mathcal{O}(n \log n)$	?	14	0	14
PSC_i	$\mathcal{O}(n \log n)$	?	14	0	14
IC	$\mathcal{O}(kn \log n)$	2	13	1	14
MF_e	$\mathcal{O}(kn \log n)$	?	13	0	13
MF+	$\mathcal{O}(n^2 S \log n)$	$13/11$	6	6	12
LPT+	$\mathcal{O}(n^2 S \log n)$	$4/3$	6	4	10
EX	$\mathcal{O}(n^3 S)$	$4/3$	5	4	9
MS	$\mathcal{O}(n^3)$	?	8	0	8
X-TMO	$\mathcal{NP}$	$13/11$	1	6	7
SS	$\mathcal{O}(n^2 S)$	?	7	0	7
KOMP	$\mathcal{O}(n^2 S)$	?	7	0	7
PSMF+	?	$13/11$	0	6	6
MSS	$\mathcal{O}(n^2 S^2)$	2	4	1	5
HI	?	$5/4$	0	5	5
DIST	$\mathcal{NP}$	$4/3$	1	4	5
MMIPMH	?	$4/3$	0	4	4
SLACK +	?	$4/3$	0	4	4
MSK	$\mathcal{O}(n^3 S^2)$	2	3	1	4
H_1	$\mathcal{O}(n^2 \log(1/\varepsilon + 1/\varepsilon^2))$	?	2	0	2
3-PHASE	?	2	0	1	1
KK	$\mathcal{NP}$	?	1	0	1

The *IC* algorithm [154] was compared with the LPT CH on instances ${}^{500}_{10}I$, using five different distributions. In these experiments, IC showed comparable but generally performed weaker than LPT. The authors also concluded that the IC algorithm is very fast for the considered set of instances.

The *KOMP* algorithm [112] was compared with LPT and MF CHs on ${}^{100}_{10}I$ instances with controlled $\frac{p_{min}}{p_{max}}$ ratio. The results clearly indicated KOMP's dominance over the

other algorithms with respect to solution quality. Additionally, MF showed slightly better performance than LPT. However, the reported computational times of KOMP are sometimes up to two orders of magnitude larger than in MF and LPT cases, raising questions about its practical usability.

The *EX* algorithm [157] was compared with the LPT CH on ${}^{60}_{5}I$ instances for ten different uniform distributions of job durations. EX performed better without a significant increase in computational time.

The *3-PHASE* algorithm [135] was compared with IC and ICII; BIN HE; and LPT CH algorithms on the F_U instances (introduced in this work). 3-PHASE showed better performance than IC, ICII, and LPT CH, where LPT CH outperformed IC and ICII. IC and ICII showed identical performance. BIN solved to optimality more instances than 3-PHASE; however, required a larger computational time.

The *X-TMO* algorithm [159] was compared with LPT, MF, and MF_e CHs on ${}^{100}_{12}I_{\mathcal{U}_{[20,250]}}$ instances. As the initial solution for X-TMO each of CHs' solutions is provided and it improved all of them with a significant increase in computational time. The results of comparing the CHs on the same instances suggested the following order: MF_e, MF, and LPT.

The *PI* algorithm [160] was compared with LPT and MF CHs on ${}^{30}_{10}I$ instances with four different uniform distributions of job processing times. The authors reported the dominance of PI over CHs with respect to solution quality and a negligible increase in computational time. Among CHs, MF performed better.

The CP_{99}algorithm [17] was compared to a simple ILP solver on ${}^{15}_{5}MK_{\mathcal{U}_{[1,100]}\mathcal{U}_{[10,100)}\mathcal{U}_{[50,100]}}$ instances, within a node limit of $5 \cdot 10^5$. Without providing the resulting table, the authors concluded that CP_{99} performs well on all instances. In the cases when the ILP solver provided optimal solutions, CP_{99} was able to find the same solutions within a significantly smaller running time. When the ILP solver failed, CP_{99} always converged to a feasible solution.

The *ME* algorithm (representing the best results of 1-SPT, 1-BPT, K-SPT, and K-BPT) [162,163] was compared with the LPT CH; DM HE (with a backtrack limit of $4 \cdot 10^3$); and the 3-PHASE algorithms on F_{NU} instances. DM dominated in this experiment, with ME outperforming the remaining algorithms, and LPT being the weakest. In the subsequent experiments, ME was compared with LPT and 3-PHASE on *BP* and *TR* instances. ME performed the best, while LPT was consistently the weakest. In [162], DM, 3-PHASE, and LPT were compared on F_U instances, where DM outperformed 3-PHASE, which in turn was better than LPT.

The *HI* algorithm [139,220] (with an iteration limit of 10^3 for the TS improvement phase) was compared with LPT CH; DM HE (with a backtrack limit of $4 \cdot 10^3$); and the 3-PHASE algorithms on F_U instances. HI produced the best results, followed by DM and 3-PHASE. On F_{NU} instances, HI was compared with LPT, DM, and ME algorithms, yielding similar results: HI dominated, DM was second, and ME was third. Detailed comparison results are provided in [251].

The *MSS* algorithm [95] was compared on the ${}^{10^4}_{15}\widetilde{C}_{1-5}$ instances with DM and BIN HE solvers. The results provided by MSS and DM were close, while MSS dominated in comparison with BIN. For *PP* instances, MSS achieved results similar to DM, where DM outperformed BIN. MSS was also compared with HI on *F* instances, showing comparable results, although sometimes slightly weaker.

The *MSK* algorithm [14] was compared with MSS and HI on *F* instances. The authors did not provide detailed comparison results, just the direct comparison by the number of better, equal, and worse solutions. They concluded that MSK performed better than the other two algorithms. MSK also showed a lower average gap compared to DM HE on ${}^{10^4}_{15}\widetilde{C}_{1-5}$ instances.

The *CA* algorithm [164] was compared with DM HE; and HI on F_U instances, where CA dominated and HI was the second. The comparison on F_{NU} instances shows that CA clearly outperformed DM, HI, and ME, solving all instances to optimality, with HI being the second, followed by DM. On *BP* instances, CA performed similarly to HI, requiring slightly larger computation time, while ME was the third and DM took the last place. On *TR* instances, CA dominated HI, DM, and ME, solving all instances to optimality. The order of remaining algorithms was: HI, ME, DM.

The KL_h algorithm [122] was compared with HI and $DIMM_{SS}$ MH on *F* instances. KL_h solved all instances to optimality within less time than the other two algorithms. The other two algorithms were not able to solve all instances, where $DIMM_{SS}$ was more successful.

The *MMIPMH* algorithm [165] outperformed the LPT CH on a very small $\frac{9}{4}I$ instance set based on a direct comparison of makespan values.

In [166], the authors compared 70 introduced IHs with each other on a subset of *Iogra* instances. The ten best-performing algorithms are identified, as it was not possible to distinguish between them with statistical significance.

The PTAS algorithms for $P||C_{max}$, as well as EE and HE algorithms, make the most sense in theoretical discussions. The primary measure for comparing them is WTC. However, some PTAS algorithms are implemented and experimentally compared with CHs.

The ε-*DUAL* algorithm, with $\varepsilon = \frac{1}{5}$, was compared with the MF_e, MF, and LPT CHs on the $\widetilde{E_{2\mathcal{N}_{450}}}$ instances in [133], as it is mentioned earlier in this section. The average performance of the ε-DUAL algorithm was close to its worst error bound, generating low-quality solutions compared to other methods. However, the required running time was comparable to others.

The *BDJR* algorithm [174], parallelized on 16 CPU cores was tested with $\varepsilon \approx 17.29\%$. The obtained results were comparable with LPT, MF, and DJMS CHs on *E* instances; however, the required computational times were significantly longer.

In Figure 9, the relationships between all heuristic algorithms are displayed, based on the available empirical results. The orange color represents IH algorithms, the light orange represents CH algorithms, and the lightest orange (at the very bottom left) represents PTAS algorithms. Additionally, small blue nodes present some exact algorithms, making the connection between heuristic and exact algorithms, i.e., between Figures 9 and 8. Dashed-line edges are used merely to highlight necessary edge intersections. The graph is divided into two major parts: the left part primarily features *F* edges, while the right side is dominated by *E* edges. However, these two parts are more connected than in Figure 8. The first connection between these two parts is established by F_{NU} edges, linking PSMF with algorithms from the *F*-part of the graph. The second connection arises from the F_U edge, connecting PSC and MF+ with the *E*-part. On the *E*-part, there is an additional *F*-path from SLACK+ to LPT. However, this path is problematic because it is based merely on better/worse comparisons, which are not informative enough to support the integration of this path into the *F*-part. It can be seen that LPT has the largest number of input edges, indicating that it is the most studied and widely used heuristic for this problem.

Once again, due to the lack of empirical comparisons in the literature, several compromises had to be made to distinguish between results across various instance sets. This may lead to some unclear relationships.

For example, considering the results on *BP* instances, the order of PSC_i, PSC, and SPS CHs should be reversed. However, as they belong to the *F*-part, the presented order is established, although, the other one is indicated too.

Similarly to the case of exact algorithms, no single heuristic method that dominates across all instances could be identified based on the available results. Therefore, it is crucial to perform a comparison of all methods on a wide range of instances under equal conditions

to gain a comprehensive understanding of the practical performance of the proposed heuristics. Due to the fact that CH algorithms are usually quite simple to implement and that they represent a good starting point for many other methods, the results of this SLR indicate that they might be extensively used in future comparisons. Additionally, special attention should be paid to selecting appropriate parameters for each particular algorithm.

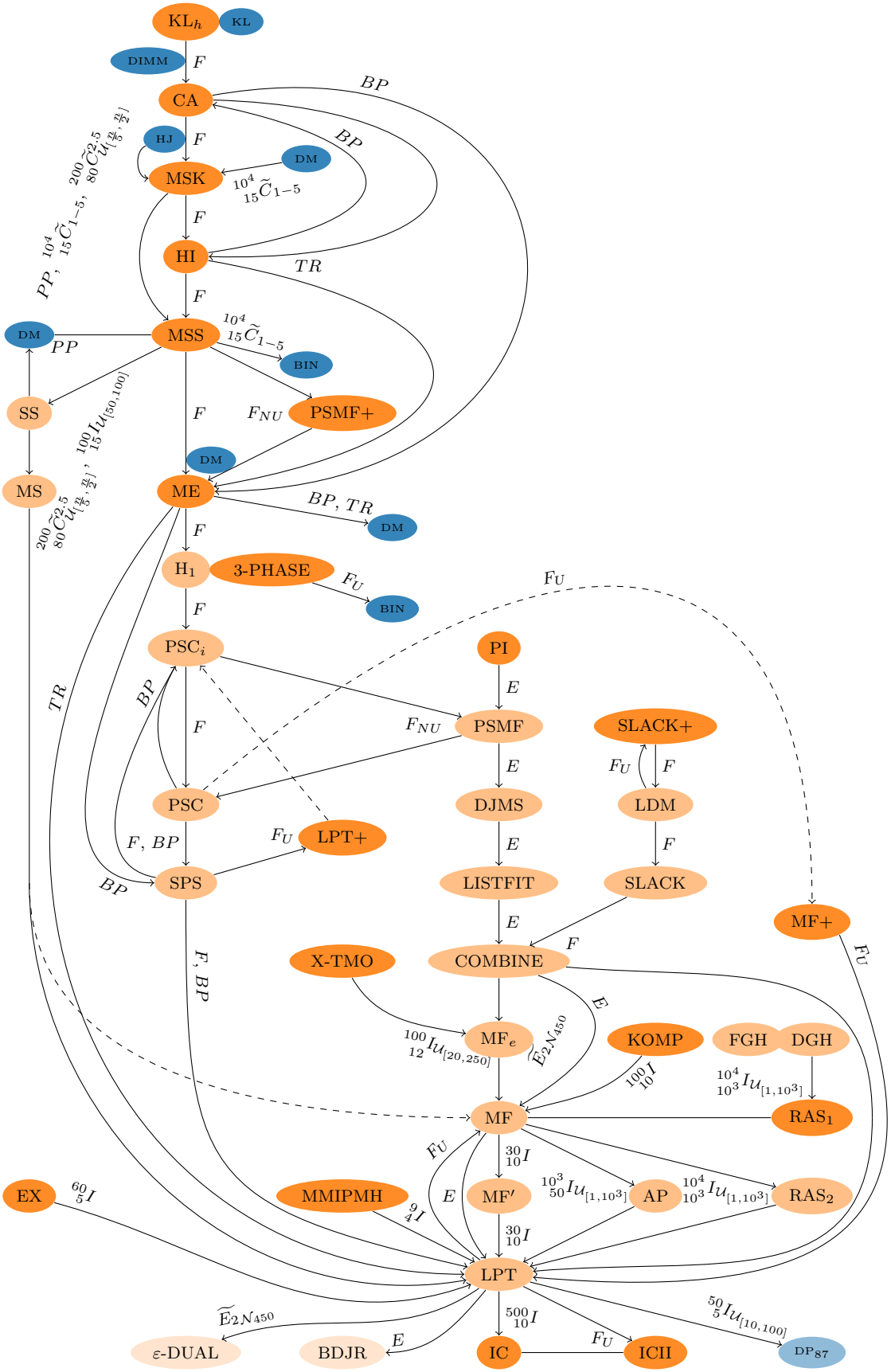

Figure 9. Graph of experimental comparisons of heuristic algorithms.

6.3.3. Metaheuristics Comparisons

Metaheuristics generally provide high-quality solutions, although without a guarantee. No improvement with respect to the AR of the initial solution was certified. An initial solution is usually generated by some CH or IH. Moreover, the performance of these algorithms heavily relies on the values of the parameters, making WTC an unreliable measure. Therefore, the performance of MHs can be estimated only by experimental evaluation.

The SA_{88} algorithm [10] was tested on a small instance set ${}^{10^3}_{12}I$ and the *G*78 instance. The experimental evaluation on ${}^{10^3}_{12}I$ instances was used to discuss the quality of the obtained solutions. The author concluded that only for $m = 2$ SA_{88} can find the optimal solutions. As the number of machines increases, finding optimal solutions becomes harder. For the *G*78 instance, based on private correspondence with its creator, the author concluded that SA_{88} outperformed several tested approximate algorithms. However, no information about the specific approximate algorithms is provided. Considering that Graham created *G*78, one can assume that LPT was certainly among them.

The *HG* algorithm [178] was tested on the ${}^{2\cdot10^3}_{50}I_{\mathcal{U}_{[0,1]}}$ set of instances with known optimal solutions. The provided results show an average relative error of at most 10^{-5}. In addition, HG algorithm is compared with SA_{88} on the *G*78 instance. In this experiment, HG provided a solution of better quality.

The SA_{97} algorithm [179] was evaluated on the ${}^{5\cdot10^3}_{50}I_{\mathcal{U}_{[10^3,5\cdot10^4]}}$ instance set. The authors examined the influence of neighborhood types and the number of iterations on the quality of the provided solution. Two types of neighborhoods are combined with two values of iteration limits (100 and 10^3). The reported gap with respect to L_2 ranged from 0.2% for small-sized instances up to 3% for larger instances. However, no comparison with other algorithms was provided.

The *TGR* algorithm [180] was compared with HG on the ${}^{2\cdot10^3}_{50}I_{\mathcal{U}_{[0,1]}}$ instance set and, in the majority of the cases, it provided better quality solutions requiring smaller number of iterations.

The GA_{99} and SA_{99} algorithms [181] were compared with each other and with the LPT CH on instance ${}^{7}_{3}I$. The results show that GA_{99} and SA_{99} are comparable and that they outperform LPT. In the next experiment, GA_{99} showed better performance than SA_{99} on instances ${}^{10}_{2}I$ and ${}^{30}_{10}I$.

The *AIS* and SA_{02} algorithms [161] were compared with each other, as well as with the LPT and MF CHs, and the LPT+ and MF+ IHs on F_U instances and five instances from [127] obtained using a stepwise distribution. This job duration distribution is known to produce the worst-case instances for LPT. In the first experiment on F_U instances, AIS performed slightly better than SA_{02}, although requiring an order of magnitude larger processing time. The heuristic algorithms were ranked in decreasing order of solution quality as follows: LPT+, MF+, LPT, and MF. In the second experiment, 25 instances ${}^{95}_{2}I_{\mathcal{U}[10^9,10^{10}]}$ from [252] were used to compare only the two metaheuristics (due to the weak performance of other algorithms). AIS was superior with respect to solution quality.

The *AntMPS* algorithm [7] was tested on ${}^{10^3}I_{\mathcal{U}[20,100]}$ instances for BPP and on *TR* instances. Interestingly, AntMPS, as a $P||C_{max}$ metaheuristic, was directly compared with different BPP metaheuristics, that were actually solving different problems on the same instances. However, AntMPS was given an optimal number of bins as the input value for m to simulate the $P||C_{max}$ problem. The resulting makespan actually represents bin capacity and, if it is not exceeded, the corresponding instance is considered as solved to optimality. With the stopping criterion limited to 10^3 iterations, AntMPS was able to solve the majority of instances to optimality, while for the remaining the resulting gap was minimal.

The SA_{06} algorithm [182] was compared with LISTFIT CH and PI IH on E instances. Given a stopping criterion of $30n$ iterations, it outperformed other algorithms with respect to the solution quality: providing the smallest gap with respect to L_1 for almost all instances. For E_4 class, SA_{06} found the larger number of optimal solutions. The second-best position was taken by PI. The execution times were not reported because SA_{06} turns out to be very fast, i.e., it completed all executions within less than 1s.

The ILS_{06} algorithm [183] was compared with the LPT CH on ${}^{10^3}_{40}I_{\mathcal{U}_{[1,100]}}$ instances, generated by the authors, where it demonstrated better performance. They used interesting performance measure of algorithms for a single instance, calculated as the ratio between the makespan produced by the algorithm for that instance and the best-known makespan for all instances of the same size. In general, ILS_{06} outperformed LPT with respect to the solution quality. However, the authors noticed different behavior of ILS_{06} algorithm related to $\frac{n}{m}$ radios, ranging from 2 to 333, where the largest improvement of 7.21% over LPT was achieved for $\frac{n}{m} \approx 3$. The authors also observed that the largest execution times correspond to the smallest $\frac{n}{m}$ ratios. ILS_{06} solved 83% of instances to optimality.

The *HDNN* algorithm [184] was compared with the LPT CH on ${}^{10}_{3}I_{\mathcal{U}_{[1,3]}}$ instances, where it outperformed LPT in reasonable time. As a performance measures, the authors used the gaps between the obtained solution and both the LPT solution and the optimal solution.

The $HDNN_i$ algorithm [185] was compared with the LPT CH on ${}^{100}_{5}I_{\mathcal{U}_{[1,3]}}$ instances, also demonstrating superior performance in reasonable time, and using the same measures as for HDNN.

The *TCNN* and $TCNN_i$ algorithms [186] were compared with HDNN on ${}^{50}_{3}I_{\mathcal{U}[1,50]}$ instances, where $TCNN_i$ showed the best performance, followed by TCNN. The authors measured the best and average makespan over 100 random instances with the same input parameter values.

The $DIMM_{SS}$ algorithm [114] was compared with 1-SPT (one of four ME algorithms) and HI IHs on F_U instances. The presented results show that $DIMM_{SS}$ provided the best results measured as the relative gap with respect to the $\max(L_3, L_{HS}, L_\theta)$. The second was HI which, in the majority of cases, required less computational time. In the second experiment, $DIMM_{SS}$ was compared with 3-PHASE, 1-SPT, and HI algorithms on F_{NU} instances, with similar results. 3-PHASE showed the worst performance.

The *MVNS* and GA_{09} algorithms [5] were compared with the LPT CH on ${}^{200}_{20}I_{\mathcal{U}[1,100]}$ instances, where MVNS showed the best results and GA_{09} performed the worst, with respect to the relative gap between produced makespan and L_1. Regarding the execution time, MVNS also outperformed GA_{09}.

The BCO_{09}algorithm [6] was tested on the $IT_{10,20,30,40,50,100}$ set of instances, achieving optimal solutions for all instances within the execution time less than 2 s for a single instance.

The VNS_{09} and MC_{09} algorithms [187] were compared on the $IT_{10,20,30,40,50,100}$ set of instances. Better performance was demonstrated by VNS_{09}, providing optimal solutions for all instances. Additionally, VNS_{09} required less CPU time than BCO_{09} in [6] .

The $DPSO_{09}$ and *HDPSO* algorithms [189] were compared with SA_{06} on E instances. The results showed that $DPSO_{09}$ slightly outperformed SA_{06}, while HDPSO performed significantly better than $DPSO_{09}$, with respect to relative gap between produced makespan and L_1. It is important to note that the results for SA_{06} on E_2 were not consistent with those reported in [182].

The *DSHS* algorithm [190], with a stopping criterion of 100 iterations, was compared with HDPSO and SA_{06} on $E_{1,2}$ instances, showing better results than both competitors with respect to both L_1-gap and execution time. As expected, SA_{06} performed the worst regarding the both measures.

The DHS_{11} and *HDHS* algorithms [191] were compared with LPT CH, GA_{99}, SA_{06}, and MVNS on ${}^{200}_{20}I_{\mathcal{U}_{[1,100]}}$ instances. All MHs were limited to 100 iterations. Both introduced algorithms outperformed the other competitors, where HDHS performed the best with respect to the quality of solutions (L_1-gap); however, requiring longer execution time than DHS_{11}. Unexpectedly, all other competitors performed worse than LPT. This was surprising because MVNS was tested on the same type of instances in [5], where it outperformed LPT. This discrepancy can be explained by the fact that experiments were performed on the newly generated test instances. However, the results for LPT seem to be copied from [5].

The *RIVNS* and *HIVNS* algorithms [192] were compared with LPT CH; GA_{99}, SA_{06}, and MVNS on ${}^{200}_{20}I_{\mathcal{U}_{[1,100]}}$ instances. All MHs were limited to 100 iterations. RIVNS and HIVNS outperformed the other competitors, with HIVNS emerging as the best, with respect to both quality of solution (L_1-gap) and execution time. The experiments were conducted using the same version of instances as in [190], and the results of both algorithms were better than those of HDHS. In this experiment, LPT achieved better average solution quality than GA_{99} and MVNS, while SA_{06} was just slightly better than LPT.

The SA_{12} algorithm [193] was compared with SA_{06} on E_2 and ${}^{500}_{20}I_{\mathcal{U}[1,100],\mathcal{U}[100,800]}$ instances, where it achieved better results with respect to L_1-gap. It is interesting to note that the authors reimplemented the SA_{06} algorithm and regenerated E_2 instances, but they were unable to reproduce the SA_{06} results and reported worse performance of SA_{06} with respect to [182].

The *SPPSO*, $DPSO_{12}$ [253], and PSO_{spv} [254] algorithms were compared in [194] on ${}^{500}_{50}I_{\mathcal{U}[1,100],\mathcal{U}[100,800]}$ instances. All algorithms used population size n and were limited to $\frac{2 \cdot 10^3}{n}$ iterations. SPPSO dominated over other algorithms with respect to average L_1-gap, followed by PSO_{spv}.

The BCO_{12} algorithm [177] was compared with a BPP solver, and in [211] with DM and DIMM HE algorithms on *IO*, IT_{500}, and *RN* instances. Both HE algorithms outperformed BCO_{12} solving all instances to optimality and having much better execution times. DM is faster than DIMM.

The DHS_{12}, *BHS*, and DHS_{LS} algorithms [195] were compared in two experiments. In the first experiment, BHS, DHS_{12}, and DHS_{LS} were compared on E_2 instances, where DHS_{LS} dominated, followed by DHS, with respect to average L_1 gap, and all algorithms had similar execution times. In the second experiment, DHS_{12}, DHS_{LS}, SA_{06}, $DPSO_{09}$, and HDPSO were compared on *E* instances. With respect to the average L_1 gap, DHS_{LS} dominated, followed by HDPSO provided slightly worse average solution quality, but within a significantly shorter execution time. DHS, SA_{06}, and $DPSO_{09}$ ranked third, fourth, and fifth, respectively, and also required significantly longer execution times. It is worth mentioning that the general conclusions, and in particular the relationship between $DPSO_{09}$ and SA_{06}, are different from those in [189].

The *CSA* algorithm [196] limited to 10^3 iterations was compared with SA_{06}, $DPSO_{09}$, and HDPSO on *E* instances, considering average gap with respect to L_1. CSA provided solutions similar to HDPSO with significantly lower CPU time consumption. Both of them outperformed SA_{06} and $DPSO_{09}$. Regarding this experiment, again, it is important to mention significant differences in the presented results compared to older studies.

The *GES* and *GES+* algorithms [197] limited to 10^3 iterations were compared with SA_{06} on *E* instances, with respect to L_1, where GES+ emerged as the best and GES ranked as the second. In addition, the new algorithms required shorter average running times. The results for SA_{06}, once again, were not consistent with older papers.

The *ICSA* algorithm [198] with limitation to 10^3 iterations, was compared with SA_{06}, $DPSO_{09}$, HDPSO, DHS_{12}, DHS_{LS}, and CSA on *E* instances. The authors reported the domination of ICSA in terms of solution quality. In the performed experimental evaluation,

the results of the older algorithms were taken from the literature, and the results of ICSA were obtained using newly generated E instances with unspecified parameters used for generation. We have a reasonable doubt about the reliability of the provided results. Based on the analysis of the results from previous experiments, the methodology of using E instances is questionable. As [198] provides the most comprehensive comparisons of different algorithms so far, it is used here for a more detailed analysis.

In the presented experiments related to E instances, for each combination of parameters, the authors randomly generated 50 problem instances without providing details about the random seeds. They presented the average gap with respect to L_1. Let us focus on the small group E_4, because of the simplicity of its instances and the possibility of solving them to optimality using a commercial linear programming solver. We generated 50 groups of E_4 instances (50 generations of 50 instances for each parameter combination), and, in the last two columns of Table 15, we present the best (OPT_B) and the worst (OPT_W) average gaps of optimal results among these 50 groups, obtained using the exact solver. The table also contains the characteristics of instances and results of other algorithms provided in [198].

As shown in Table 15, the average results of all metaheuristics, except ICSA, are between OPT_B and OPT_W. If we assume that all presented heuristics provided near-optimal solutions for all instances from E_4, these results can be considered acceptable. However, the average gap with respect to all instances presented for ICSA is significantly smaller than the best average gap of OPT_B, leading us to conclude that the authors of [198] may have been particularly fortunate when generating the E instances.

Table 15. OPT_B, OPT_W and results of metaheuristic solvers presented in [198] for E_4 instances. Bolded values indicate identified discrepancy.

m,*n*	*D*	SA_{06}	$DPSO_{09}$	HDPSO	DHS_{12}	DHS_{LS}	CSA	ICSA	OPT_B	OPT_W
2,9	$\mathcal{U}_{[1,20]}$	1.001	1.000	1.000	N/A	N/A	1.000	1.000	1.000	1.001
	$\mathcal{U}_{[20,50]}$	1.001	1.001	1.001	N/A	N/A	1.001	1.001	1.002	1.001
	$\mathcal{U}_{[50,100]}$	1.004	1.004	1.004	N/A	N/A	1.004	1.003	1.002	1.003
	$\mathcal{U}_{[100,200]}$	1.004	1.004	1.004	N/A	N/A	1.004	1.004	1.002	1.005
	$\mathcal{U}_{[100,800]}$	1.002	1.002	1.002	N/A	N/A	1.002	1.001	1.002	1.003
3,10	$\mathcal{U}_{[1,20]}$	1.001	1.001	1.001	N/A	N/A	1.001	1.002	1.003	1.006
	$\mathcal{U}_{[20,50]}$	1.008	1.007	1.007	N/A	N/A	1.007	1.005	1.006	1.009
	$\mathcal{U}_{[50,100]}$	1.010	1.009	1.009	N/A	N/A	1.010	1.006	1.011	1.010
	$\mathcal{U}_{[100,200]}$	1.017	1.016	1.016	N/A	N/A	1.016	1.008	1.010	1.020
	$\mathcal{U}_{[100,800]}$	1.009	1.009	1.009	N/A	N/A	1.009	1.006	1.007	1.008
Average		1.0057	1.0053	1.0053	1.0064	1.0061	1.0054	1.0036	**1.0046**	**1.0067**

The $HIVNS_1$, $RIVNS_1$, $HIVNS_2$, and $RIVNS_2$ algorithms [199] limited to 10^3 iterations, were compared with GA_{99}, SA_{06}, MVNS, RIVNS, and HIVNS on ${}^{200}_{20}I_{\mathcal{U}_{[1,100]}}$ instances. The authors generated ten instances for the same combination of parameters and used the gap with respect to L_1 as the performance measure. All presented algorithms outperformed the older competitors. $RIVNS_1$ and $RIVNS_2$ achieved the best average results, while $RIVNS_2$ was slightly faster. $HIVNS_1$ performed the worst.

The GWO_{19} and GA_{19} algorithms [200] were compared with each other on E instances, where GWO_{19} outperformed GA_{19}. However, the type of measurement used was not explained, and therefore, the results were unusable for additional comparisons.

Figure 10 illustrates the relationships between the employed metaheuristic algorithms based on the available empirical results. MHs from the P-class are represented in dark green, while light green is used for S-class MHs. Additionally, some exact and heuristic algorithms (in the corresponding colored nodes) are included to establish a connection with the graphs in Figure 8 and Figure 9. As the first observation, it is evident that the graph can be separated into several parts. The largest (central) part is primarily characterized by the utilization of E instances. Based on our analysis of the performances of the ICSA

algorithm [198], we demonstrated that this branch should not be considered as a fully accurate representation of the relationships between the corresponding algorithms.

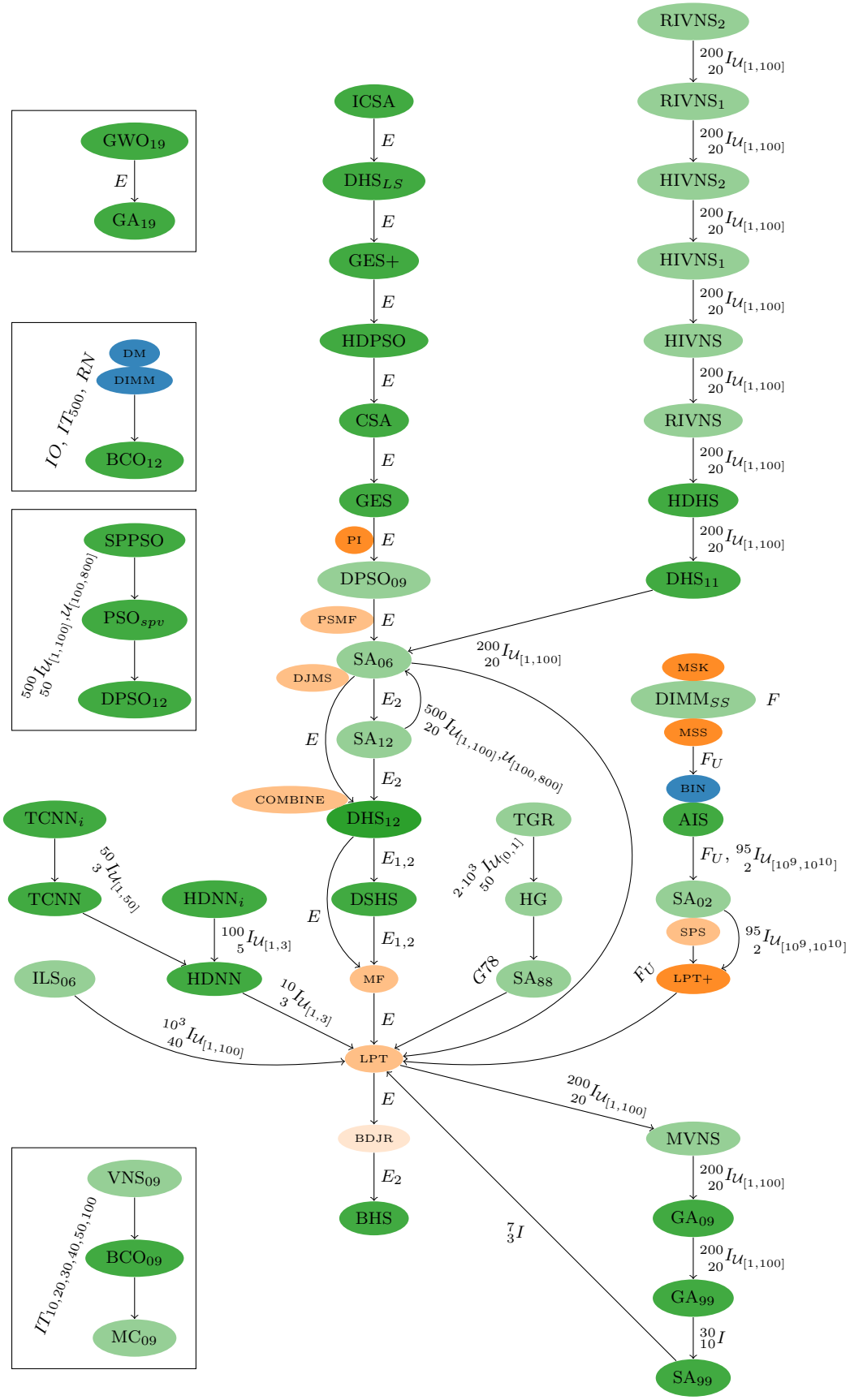

Figure 10. Graph of experimental comparisons of metaheuristic algorithms.

The second (upper right) part of the graph contains algorithms compared on ${}^{200}_{20}I_{\mathcal{U}_{[1,100]}}$ instances and the relationships between the algorithms could clearly be established (algorithms performing better are on top). Additionally, there is a smaller, central right, part of the graph that involves F_{UI} instances, which helps to establish relationships between metaheuristics and the results of exact and heuristic algorithms. ANN-based metaheuristics are presented in the left part of the graph, besides ILS_{06}. Finally, in isolated parts of the graph, comparisons of algorithms that have no direct relationship with other algorithms provided. Similarly to heuristic and exact methods, no single metaheuristic method dominates across all instances.

7. RQ4: Discussion, Challenges, and Future Work

Although the $P||C_{max}$ problem was introduced more than half a century ago, Figure 11 shows that interest in publishing research related to $P||C_{max}$ did not diminish. Several papers have been published on this topic every year. We observe that majority of publications related to metaheuristic approaches were published from the early 2000s to the 2020s. On the contrary the publications on heuristics and exact solvers, which exhibit a consistent developmental trend from the inception of the problem.

The identified limitations of any SLR are related to biases in availability of publications and in the study selection processes, to inaccuracy in the study extraction process, and to misclassification of published results [255]. We have identified the following challenges, which provide the exciting opportunities for future work:

1. Isomorphic versions of $P||C_{max}$ problem;
2. A lot of taxonomies available;
3. Standardization of identified groups of instances;
4. Instance nomenclature limitations;
5. Fair algorithm performance evaluation;
6. The state-of-the-art algorithm identification.

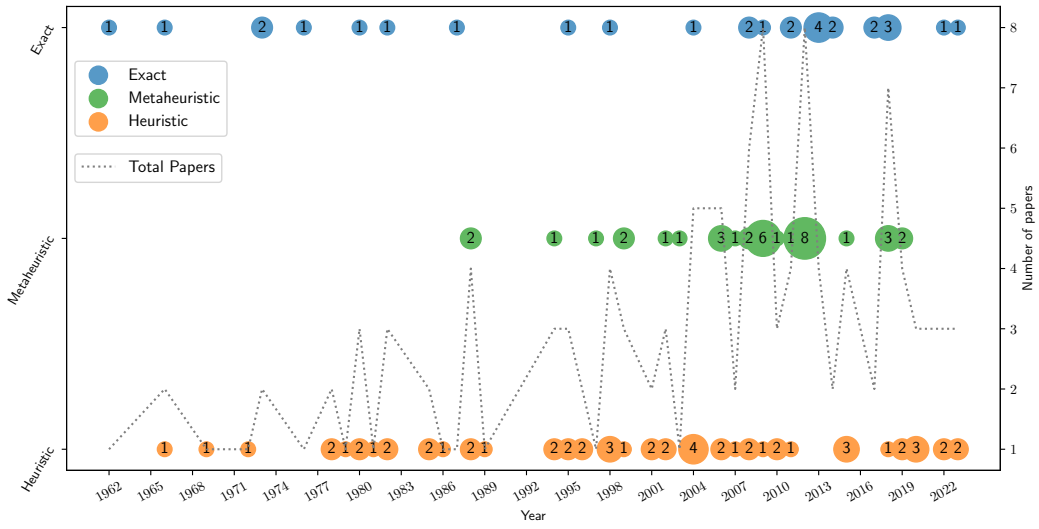

Figure 11. Published solving methods per year.

Isomorphic versions and multiple taxonomies challenges could be addressed by identifying and presenting various isomorphic versions of $P||C_{max}$ problem and selecting the taxonomy that was most appropriate for the corresponding studies. However, there might still be cases of specialized $P||C_{max}$ problem versions as well as algorithms that were not covered by our exhaustive search. An example is a set of exact solvers and approximate algorithms for high-multiplicity instances, i.e., instances containing a small

number of groups with a lot of identical jobs [256–259]. Theoretical verification of model equivalence and performance evaluation of corresponding algorithms for each newly discovered isomorphism represent promising a avenue for future research.

Instance groups standardization and instance nomenclature allow us to establish benchmarks and simplify the procedure of checking and referencing published results. To that end, we have constructed a graph of algorithm comparisons for each group of algorithms in the selected studies. This graph from each group of the algorithms shows that it is easy to see the relationships between them in reference to testing on the same instance groups. However, the facts that not all instances can be considered as standardized and not all algorithms were tested on all instances resulted in our inability to provide a comprehensive graphical representation. Our nomenclature covers four characteristics of instances, which to the best of our knowledge is the most efficient way to describe all instances in the instance groups we identified. The comprehensive set of standardized instances still needs to be identified and systematically characterized.

Fair algorithm performance evaluation challenge is rather a group of many inherent issues that include but are not limited to using ad-hoc instances, not providing the full set of characteristics of the results, instances, solvers, controlling seed values for random variables, and the lack of public repositories containing the algorithms, instances and their optimal/best-known solutions. By making at least the standardized groups of instances available, along with all the identified characteristics, we are setting the stage for fair evaluation of algorithm performance. Resolving the remaining issues remains as a potential subject for future work.

In addition, fair algorithm comparison must include the analysis of all involved components. From the selected studies, we learned that the choice of the lower bounds affects both the solution quality and the speed of obtaining the solution. Thus, we attempted to exhaustively explain all available lower-bounding strategies. Moreover, many algorithms and their performance depend on the underlying algorithm that was implemented to help in the solution finding process. We identified and described many such algorithms. Combining various parts from different algorithms should be additionally explored, as it may result in the development of more efficient search procedures.

Another issue for fair algorithm performance evaluation is related to testing stochastic algorithms. Here, it is important to determine the number of repetitions (executions, runs) of the algorithm by combining minimal statistical requirements and a number of runs already utilized in the literature. For example, the authors of [260] suggest that any number of runs between 20 to 100 could be sufficient for a good estimation of the algorithm's performance, in [137] the authors used 100 repetitions, while in [12], 30 repetitions were performed. Standardization of the number of repetitions, supported by theoretical statistical considerations, still needs to be performed.

For some CH and IH algorithms, only AR and WTC parameters were available. Even though this was not a primary focus of our research, we have tried to establish AR and WTC for the remaining CH and IH algorithms to be able to include all of them into comparison. As a future work we envision the systematic experimental evaluation of all CH and IH algorithms on the standardized set of instances, under the same condition.

The state-of-the-art algorithm identification is directly tied with fair algorithm performance evaluation. Due to the fact that not all algorithms were tested on all instances, under the same conditions, and other related issues, we were unable to identify even the best-performing algorithm for any of the standardized groups of instances. Instead, we have established the standardized framework to characterize the state-of-the-art candidates, i.e., the algorithms that perform the best on the majority of instances from all standardized groups. Another interesting utilization of the obtained results involves the selection of

the best-performing algorithm for the unseen instances. More precisely, future work may include building a rich database of experimental results for different algorithms on different sets of instances. Such a database would enable AI/ML techniques to predict the difficulty of any new instance that has not been tested yet [1].

8. Conclusions

We conducted a systematic literature review of optimization algorithms applied to the $P||C_{max}$ problem. We summarized and categorized the "state-of-the-art" methods, benchmark test instances, and performance indicators. As a result, we identified a taxonomy of optimization algorithms where the categories represent high-level types of optimization methods (exact, heuristic, metaheuristic) that have been used in the attempts to solve the $P||C_{max}$ problem.

Researchers are actively working on developing new methods that need to be evaluated appropriately. We have applied a systematic literature review mapping principles to try to find them all and provided statistics of all the identified literature along with the challenges and their resolutions. One of the problems we faced was the symmetry among the optimization problems that are similar to $P||C_{max}$. Therefore, the $P||C_{max}$ problem was recognized under different names, not only in various communities that tried to solve it but even within the communities. In was very challenging and time consuming to identify all these symmetric problems as $P||C_{max}$. Another difficulty was related to test instances. Several dozens of various sets of instances were used in different papers, thus making difficult to obtain fair comparisons of the used methods. We have standardized four main groups of instances and made them publicly available along with explanations of their characteristics. Various criteria (performance indicators) had been used to compare optimization algorithms. We have identified a considerable number of them that were applied to the $P||C_{max}$ problem. We made a proper match between categories of methods and performance indicators, and, for each category, we proposed a suitable subset of indicators that should be enough to objectively evaluate the newly developed optimization method.

The first avenue for future research could be identifying the best algorithms (the-state-of-the-art) for $P||C_{max}$ problem. This could be performed by testing best solvers from all categories on all standardized instances under the same conditions. Then, the best-performing algorithms should be evaluated against each other and a set containing the-state-of-the-art methods should be formed. It is not realistic to assume that a single best approach with respect to all instances could be found. Upon an in-depth experimental analysis, potentially supported by the theoretical consideration, the best that can be carried out is to determine the combination of algorithms' and instances' characteristics that yield a good performance. As the second avenue of future studies we envision the application of artificial intelligence techniques, machine learning in particular, to predict the performance of a given algorithm on some newly introduced instances. The third avenue could be the hybridization of algorithms from different categories to build a solver capable of efficiently solving a wide variety of instances. The fourth avenue could be the generation of additional sets containing standardized instances for the $P||C_{max}$ problem. Moreover, theoretical analysis of the problem and the development of new solvers could be another avenue of future research.

In addition, our considerations could be generalized as a framework to investigate other optimization problems. A fair comparison of optimization methods, as well as rich data for learning and predicting the performance of various algorithms, would be of great importance and applicability in many domains, including rather exotic blockchain maintenance and security evaluation. However, significant work is required to prepare all relevant data and the results for the application of the considered algorithms to each particular

optimization problem instances. Automating this process may also be an interesting avenue for future research.

Author Contributions: Conceptualization, T.D., D.O. and D.R.; methodology, D.O., J.K. and D.R.; software, D.O, R.D., A.U. and M.J.; validation, D.O., R.D., A.U., M.J., T.D., T.J.K. and D.R.; formal analysis, T.D. and D.O.; investigation, R.D., A.U., M.J. and D.O.; resources, D.O., T.J.K. and D.R.; data curation, R.D., A.U., M.J. and D.O.; writing—original draft preparation, D.O., R.D., A.U., M.J., T.D., T.J.K. and D.R.; writing—review and editing, D.O., R.D., A.U., M.J., T.D., T.J.K., J.K. and D.R.; visualization, D.O., R.D., M.J. and D.O.; supervision, T.D. and D.R.; project administration, T.D. and D.R.; funding acquisition, T.D. and D.R. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by Penn State Great Valley and by the Ministry of Science, Technological Development and Innovations of Republic of Serbia, agreements nos. 451-03-47/2023-01/200029 and 451-03-47/2023-01/200122.

Data Availability Statement: Project repository: https://gitlab.com/pcmax-problem/pcmax-instances. All other data could be recovered from the article.

Acknowledgments: This project has been supported by Penn State Great Valley. We would like to thank Rajan, Ram Prakassh Chinnakonda and Sudarshan, Rashmi for their help with identifying and explaining the SLR work. We thank Gharbi, Anis and Bamatraf, Khaled for providing iAF results, Mehdi, Mrad and Nizar, Souayah for providing code and instances used in their ArcFlow paper, Maxence Delorme for providing code, instructions, and helpful guidance regarding their paper. We also thank Zarges, Christine for help in standardizing *B* instances.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations and acronyms are used in this manuscript:

Abbreviation	Meaning	Group
AF	Arc-flow	Exact
B&B	Branch and Bound	
DP	Dynamic Programming	
EE	Exponential Exact	
EN	Enumeration Approach	
FPT	Fixed Parameter Tractable	
HE	Hybrid Exact	
LDE	Linear Diophantine Equations	
SSM	Sort and Search Method	
CH	Constructive Heuristic	Heuristic
IH	Improvement Heuristic	
PTAS	Polynomial Time Approximation Scheme	
ACO	Ant Colony Optimisation	Metaheuristic
ANN	Artificial Neural Network	
BCO	Bee Colony Optimization	
CS	Cuckoo Search	
GA	Genetic Algorithm	
GES	Grouping Evolutionary Strategy	
GWO	Grey Wolf Optimiser	
HS	Harmony Search	
IBA	Immune-Based Approach	
ILS	Iterated Local Search	
MC	Monte Carlo	
MH	Metaheuristic	
PSO	Particle Swarm Optimization	
SS	Scatter Search	
SA	Simulated Annealing	
TS	Tabu Search	
VNS	Variable Neighborhood Search	
BPP	Bin Packing Problem	Problems
IPMS	Identical Parallel Machine Scheduling Problem ($P\|\|C_{max}$)	
KP	Knapsack Problem	
MKP	Multiple Knapsack Problem	
MMBPP	Min-Max Bin Packing Problem ($P\|\|C_{max}$)	
MSSP	Multiple Subset-Sum Problem	
MWNP	Multi-Way Number Partitioning Problem ($P\|\|C_{max}$)	
SSP	Subset-Sum Problem	
AI	Artificial Intelligence	Terminology
BC	Blockchain	
LB	Lower Bound	
ML	Machine Learning	
OR	Operational Research	
SLR	Systematic Literature Review	

References

1. Maleš, U.; Ramljak, D.; Jakšić-Krüger, T.; Davidović, T.; Ostojić, D.; Haridas, A. Controlling the Difficulty of Combinatorial Optimization Problems for Fair Proof-of-Useful-Work-Based Blockchain Consensus Protocol. *Symmetry* **2023**, *15*, 140.
2. Garey, M.R.; Johnson, D.S. *Computers and Intractability*; Freeman and Company: San Francisco, CA, USA, 1979.
3. Pinedo, M.L. *Scheduling: Theory, Algorithms, and Systems*, 5th ed.; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2016.
4. Graham, R.L.; Lawler, E.L.; Lenstra, J.K.; Kan, A.R. Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of Discrete Mathematics*; Elsevier: Amsterdam, The Netherlands, 1979; Volume 5, pp. 287–326.
5. Sevkli, M.; Uysal, H. A modified variable neighborhood search for minimizing the makespan on identical parallel machines. In Proceedings of the 2009 International Conference on Computers & Industrial Engineering, Troyes, France, 6–9 July 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 108–111.
6. Davidovic, T.; Selmic, M.; Teodorovic, D. Scheduling independent tasks: Bee colony optimization approach. In Proceedings of the 2009 17th Mediterranean Conference on Control and Automation, Thessaloniki, Greece, 24–26 June 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 1020–1025.
7. Ritchie, G. Static Multi-Processor Scheduling with Ant Colony Optimisation & Local Search. Ph.D. Thesis, School of Informatics, University of Edinburgh, Edinburgh, Scotland, 2003.
8. Lawrinenko, A. Identical Parallel Machine Scheduling Problems: Structural Patterns, Bounding Techniques and Solution Procedures. Ph.D. Thesis, Friedrich-Schiller-Universität Jena, Jena, Germany, 2017.
9. Korf, R.E. A complete anytime algorithm for number partitioning. *Artif. Intell.* **1998**, *106*, 181–203.
10. Kämpke, T. Simulated annealing: Use of a new tool in bin packing. *Ann. Oper. Res.* **1988**, *16*, 327–332.
11. Scholl, A.; Klein, R.; Jürgens, C. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Comput. Oper. Res.* **1997**, *24*, 627–645.
12. Ostojić, D.; Urošević, A.; Davidović, T.; Jakšić-Krüger, T.; Ramljak, D. Decomposition-based efficient heuristic for scheduling. In Proceedings of the 50th International Symposium on Operational Research (SYM-OP-IS 2023), Mount Tara, Serbia, 18–21 September 2023; pp. 1027–1034.
13. Hayes, B. Computing science: The easiest hard problem. *Am. Sci.* **2002**, *90*, 113–117.
14. Haouari, M.; Jemmali, M. Tight bounds for the identical parallel machine-scheduling problem: Part II. *Int. Trans. Oper. Res.* **2008**, *15*, 19–34.
15. Brackin, M.; Jakšić-Krüger, T. Statistical considerations about modeling performance of exact solvers on problem instances of P||CMAX. In Proceedings of the 50th International Symposium on Operational Research, SYMOPIS 2023, Mount Tara, Serbia, 18–21 September 2023.
16. McNaughton, R. Scheduling with deadlines and loss functions. *Manag. Sci.* **1959**, *6*, 1–12.
17. Mokotoff, E. Scheduling to minimize the makespan on identical parallel Machines: An LP-based algorithm. *Investig. Oper.* **1999**, *8*, 97107.
18. Lenstra, J.K.; Rinnooy Kan, A. An introduction to multiprocessor scheduling. *Qüestiió* **1981**, *5*, 1981.
19. Talbi, E.G. *Metaheuristics: From Design to Implementation*; John Wiley & Sons: Hoboken, NJ, USA, 2009.
20. Iori, M.; Martello, S. Scatter search algorithms for identical parallel machine scheduling problems. In *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*; Springer Berlin Heidelberg, 2008; pp. 41–59.
21. Baker, K.R. Procedures for sequencing tasks with one resource type. *Int. J. Prod. Res.* **1973**, *11*, 125–138.
22. Baker, K. *Introduction to Sequencing and Scheduling*; Wiley, 1974.
23. Panwalkar, S.S.; Iskander, W. A survey of scheduling rules. *Oper. Res.* **1977**, *25*, 45–61.
24. Langston, M.A. *Processor Scheduling with Improved Heuristic Algorithms*; Texas A&M University: College Station, TX, USA, 1981.
25. Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A. Recent developments in deterministic sequencing and scheduling: a survey. In Proceedings of the Deterministic and Stochastic Scheduling: Proceedings of the NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems held in Durham, England, 6–17 July 1981; Springer: Berlin/Heidelberg, Germany, 1982; pp. 35–73.
26. Błażewicz, J. Selected Topics in Scheduling Theory. In *North-Holland Mathematics Studies*; Elsevier: Amsterdam, The Netherlands, 1987; Volume 132, pp. 1–59.
27. Langston, M.A. A study of composite heuristic algorithms. *J. Oper. Res. Soc.* **1987**, *38*, 539–544.
28. Sarkar, V. Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 1987.
29. Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A.H.; Shmoys, D.B. Sequencing and scheduling: Algorithms and complexity. In *Logistics of Production and Inventory*; Handbooks in Operations Research and Management Science; Elsevier: Amsterdam, The Netherlands, 1993; Volume 4, pp. 445–522.

30. Błażewicz, J.; Ecker, K.; Pesch, E.; Schmidt, G.; Weglarz, J. *Scheduling in Computer and Manufacturing Systems*; Springer: Berlin/Heidelberg, Germany, 1994.
31. Hoogeveen, J.; Lenstra, J.; van de Velde, S. Sequencing and scheduling. In *Annotated Bibliographies in Combinatorial Optimization*; Wiley: Hoboken, NJ, USA, 1997; pp. 180–197.
32. Chen, B.; Potts, C.N.; Woeginger, G.J. A Review of Machine Scheduling: Complexity, Algorithms and Approximability. In *Handbook of Combinatorial Optimization: Volume 1–3*; Du, D.Z., Pardalos, P.M., Eds.; Springer US: Boston, MA, USA, 1998; pp. 1493–1641.
33. Karger, D.R.; Stein, C.; Wein, J. Scheduling algorithms. In *Algorithms and Theory of Computation Handbook*; Atallah, M., Ed.; CRC Press: Boca Raton, FL, 1999; Volume 1, pp. 20–20.
34. Mokotoff, E. Parallel machine scheduling problems: A survey. *Asia-Pac. J. Oper. Res.* **2001**, *18*, 193.
35. Eiselt, H.A.; Sandblom, C.L. *Decision Analysis, Location Models, and Scheduling Problems*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2013.
36. Leung, J.Y. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*; CRC Press: Boca Raton, FL, USA, 2004.
37. Potts, C.N.; Strusevich, V.A. Fifty years of scheduling: A survey of milestones. *J. Oper. Res. Soc.* **2009**, *60*, S41–S68.
38. Behera, D.K. Complexity on parallel machine scheduling: A review. In Proceedings of the Emerging Trends in Science, Engineering and Technology, International Conference, INCOSET, Tiruchirappalli, Tamilnadu, India, 13–14 December 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 373–381.
39. Baker, K.R.; Trietsch, D. *Principles of Sequencing and Scheduling*; John Wiley & Sons: Hoboken, NJ, USA, 2013.
40. Schreiber, E.L. Optimal Multi-Way Number Partitioning. Ph.D. Thesis, University of California Los Angeles, Los Angeles, CA, USA, 2014.
41. Laha, D.; Behera, D.K. A comprehensive review and evaluation of LPT, MULTIFIT, COMBINE and LISTFIT for scheduling identical parallel machines. *Int. J. Inf. Commun. Technol.* **2017**, *11*, 151–165.
42. Schreiber, E.L.; Korf, R.E.; Moffitt, M.D. Optimal multi-way number partitioning. *J. ACM (JACM)* **2018**, *65*, 1–61.
43. T'kindt, V.; Della Croce, F.; Liedloff, M. Moderate exponential-time algorithms for scheduling problems. *4OR* **2022**, *20*, 533–566.
44. Deppert, M. Algorithms for Scheduling Problems and Integer Programming. Ph.D. Thesis, Universitätsbibliothek Kiel, Kiel, Germany, 2022.
45. Schryen, G.; Sperling, M. Literature reviews in operations research: A new taxonomy and a meta review. *Comput. Oper. Res.* **2023**, *157*, 106269.
46. Keele, S.; *Guidelines for Performing Systematic Literature Reviews in Software Engineering*; Technical Report; Ver. 2.3, EBSE Technical Report. vol. 5, EBSE; 2007.
47. Hu, T.C. Parallel Sequencing and Assembly Line Problems. *Oper. Res.* **1961**, *9*, 841–848.
48. Karp, R.M. *Reducibility Among Combinatorial Problems*; Springer: Berlin/Heidelberg, Germany, 2010.
49. Garey, M.R.; Graham, R.L.; Johnson, D.S. Performance Guarantees for Scheduling Algorithms. *Oper. Res.* **1978**, *26*, 3–21.
50. Lenstra, J.K.; Rinnooy Kan, A. Computational complexity of discrete optimization problems. In *Annals of Discrete Mathematics*; Elsevier: Amsterdam, The Netherlands, 1979; Volume 4, pp. 121–140.
51. Bruno, J.; Downey, P.; Frederickson, G.N. Sequencing tasks with exponential service times to minimize the expected flow time or makespan. *J. ACM (JACM)* **1981**, *28*, 100–113.
52. Achugbue, J.O.; Chin, F.Y. Bounds on schedules for independent tasks with similar execution times. *J. ACM (JACM)* **1981**, *28*, 81–99.
53. Karmarkar, N.; Karp, R.M. *The Differencing Method of Set Partitioning*; Computer Science Division (EECS), University of California Berkeley: Berkeley, CA, USA, 1982.
54. Hochbaum, D.S.; Shmoys, D.B. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM (JACM)* **1987**, *34*, 144–162.
55. Dror, M.; Stern, H.I.; Lenstra, J.K. Parallel machine scheduling: Processing rates dependent on number of jobs in operation. *Manag. Sci.* **1987**, *33*, 1001–1009.
56. Weiss, G. On almost optimal priority rules for preemptive scheduling of stochastic jobs on parallel machines. *Adv. Appl. Probab.* **1995**, *27*, 821–839.
57. Dell'Amico, M.; Martello, S. Optimal Scheduling of Tasks on Identical Parallel Processors. *ORSA J. Comput.* **1995**, *7*, 191–200.
58. Alon, N.; Azar, Y.; Woeginger, G.J.; Yadid, T. Approximation schemes for scheduling on parallel machines. *J. Sched.* **1998**, *1*, 55–66.
59. Schuurman, P.; Vredeveld, T. Performance guarantees of local search for multiprocessor scheduling. *INFORMS J. Comput.* **2007**, *19*, 52–63.
60. Hurkens, C.A.; Vredeveld, T. Local search for multiprocessor scheduling: How many moves does it take to a local optimum? *Oper. Res. Lett.* **2003**, *31*, 137–141.
61. Brueggemann, T.; Hurink, J.L.; Vredeveld, T.; Woeginger, G.J. Very large-scale neighborhoods with performance guarantees for minimizing makespan on parallel machines. In Proceedings of the International Workshop on Approximation and Online Algorithms, Eilat, Israel, 11–12 October 2007; Springer: Berlin/Heidelberg, Germany, 2007; pp. 41–54.

62. Mnich, M.; Wiese, A. Scheduling and fixed-parameter tractability. *Math. Program.* **2015**, *154*, 533–562.
63. Walter, R.; Lawrinenko, A. A characterization of optimal multiprocessor schedules and new dominance rules. *J. Comb. Optim.* **2020**, *40*, 876–900.
64. Chen, L.; Jansen, K.; Zhang, G. On the optimality of exact and approximation algorithms for scheduling problems. *J. Comput. Syst. Sci.* **2018**, *96*, 1–32.
65. Brucker, P. Scheduling algorithms. *J.-Oper. Res. Soc.* **2007**, *50*, 774–774.
66. Walter, R. Analyzing Various Aspects of Scheduling Independent Jobs on Identical Machines. Ph.D. Thesis, Friedrich Schiller University Jena, Jena, Germany, 2010.
67. Anderson, E.J.; Glass, C.A.; Potts, C.N., Machine scheduling. In *Local Search in Combinatorial Optimization*; Princeton University Press: Princeton, NJ, USA, 2003; pp. 361–414.
68. Błażewicz, J.; Dror, M.; Weglarz, J. Mathematical programming formulations for machine scheduling: A survey. *Eur. J. Oper. Res.* **1991**, *51*, 283–300.
69. Cheng, T.; Sin, C. A state-of-the-art review of parallel-machine scheduling research. *Eur. J. Oper. Res.* **1990**, *47*, 271–292.
70. Haupt, R. A survey of priority rule-based scheduling. *Operations-Research-Spektrum* **1989**, *11*, 3–16.
71. Graham, R.L. Combinatorial scheduling theory. In *Mathematics Today Twelve Informal Essays*; Springer: Berlin/Heidelberg, Germany, 1978; pp. 183–211.
72. Lenstra, J.K.; Kan, A.R. New directions in scheduling theory. *Oper. Res. Lett.* **1984**, *2*, 255–259.
73. Lenstra, J.K.; Kan, A.R.; Brucker, P. Complexity of machine scheduling problems. In *Annals of discrete mathematics*; Elsevier: Amsterdam, The Netherlands, 1977; Volume 1, pp. 343–362.
74. Gonzalez, M.J., Jr. Deterministic processor scheduling. *ACM Comput. Surv. (CSUR)* **1977**, *9*, 173–204.
75. Sahni, S. General techniques for combinatorial approximation. *Oper. Research* **1977**, *25*, 920–936.
76. Coffman, E.G., Jr.; Bruno, J.L. *Computer and Job-Shop sScheduling Theory*; Wiley: Hoboken, NJ, USA, 1976.
77. Parker, R.G. *Deterministic Scheduling Theory*; CRC Press: Boca Raton, FL, USA, 1996.
78. Bauke, H.; Mertens, S.; Engel, A. Phase transition in multiprocessor scheduling. *Phys. Rev. Lett.* **2003**, *90*, 158701.
79. Korf, R. Objective functions for multi-way number partitioning. In Proceedings of the International Symposium on Combinatorial Search, Atlanta, GA, USA, 8–10 July 2010; Volume 1, pp. 71–72.
80. Boxma, O. A probabilistic analysis of the LPT scheduling rule. In Proceedings of the International Symposium on Computer Performance Modelling, Measurement and Evaluation, San Francisco, CA, USA, 14 December 1984; Gelenbe, E., Ed.; North-Holland Publishing Company, Amsterdam, The Netherlands, 1984; pp. 475–490.
81. Boxma, O.J. A probabilistic analysis of multiprocessor list scheduling: The erlang case. *Commun. Stat. Stoch. Model.* **1985**, *1*, 209–220.
82. Coffman, E.G., Jr.; Flatto, L.; Lueker, G.S. Expected makespans for largest-first multiprocessor scheduling. In Proceedings of the International Symposium on Computer Performance Modelling, Measurement and Evaluation, San Francisco, CA, USA, December 1984; pp. 491–506.
83. Coffman, E.G., Jr.; Frederickson, G.N.; Lueker, G.S. A note on expected makespans for largest-first sequences of independent tasks on two processors. *Math. Oper. Res.* **1984**, *9*, 260–266.
84. Coffman, E.G., Jr.; Johnson, D.S.; Lueker, G.S.; Shor, P.W. Probabilistic analysis of packing and related partitioning problems. *Stat. Sci.* **1993**, *8*, 40–47.
85. Lueker, G.S. A note on the average-case behavior of a simple differencing method for partitioning. *Oper. Res. Lett.* **1987**, *6*, 285–287.
86. Frenk, J.; Rinnooy Kan, A. The rate of convergence to optimality of the LPT rule. *Discret. Appl. Math.* **1986**, *14*, 187–197.
87. Gent, I.P.; Walsh, T. Phase transitions and annealed theories: Number partitioning as a case study. In Proceedings of the European Conference on Artificial Intelligence, Budapest, Hungary, 11–16 August 1996; pp. 170–174.
88. Gent, I.P.; Walsh, T. Analysis of heuristics for number partitioning. *Comput. Intell.* **1998**, *14*, 430–451.
89. Vredeveld, T. Combinatorial Approximation Algorithms: Guaranteed Versus Experimental Performance. Ph.D. Thesis, 1 (Tesearch TU/e / Graduation TU/e), Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 2002.
90. Brueggemann, T.; Hurink, J.L.; Vredeveld, T.; Woeginger, G.J. Exponential size neighborhoods for makespan minimization scheduling. *Nav. Res. Logist. (NRL)* **2011**, *58*, 795–803.
91. Pittel, B. Perfect partitions of a random set of integers. *arXiv preprint* **2022**, arXiv:2210.00656.
92. Bauke, H.; Franz, S.; Mertens, S. Number partitioning as a random energy model. *J. Stat. Mech. Theory Exp.* **2004**, *2004*, P04003.
93. Bovier, A.; Kurkova, I. Poisson convergence in the restricted k-partitioning problem. *Random Struct. Algorithms* **2007**, *30*, 505–531.
94. Pan, A. Random Walks, Number Partitioning, and Regular Graphs. Ph.D. Thesis, The Ohio State University, Columbus, OH, USA, 2024.
95. Haouari, M.; Gharbi, A.; Jemmali, M. Tight bounds for the identical parallel machine scheduling problem. *Int. Trans. Oper. Res.* **2006**, *13*, 529–548.

96. Fekete, S.P.; Schepers, J. New classes of fast lower bounds for bin packing problems. *Math. Program.* **2001**, *91*, 11–31.
97. Webster, S. A general lower bound for the makespan problem. *Eur. J. Oper. Res.* **1996**, *89*, 516–524.
98. Held, M.; Karp, R.M. A Dynamic Programming Approach to Sequencing Problems. *J. Soc. Ind. Appl. Math.* **1962**, *10*, 196–210.
99. Fomin, F.V.; Kratsch, D. *Exact Exponential Algorithms*; Springer: Berlin/Heidelberg, Germany, 2010.
100. Cygan, M.; Fomin, F.V.; Kowalik, Ł.; Lokshtanov, D.; Marx, D.; Pilipczuk, M.; Pilipczuk, M.; Saurabh, S. *Parameterized Algorithms*; Springer: Berlin/Heidelberg, Germany, 2015; Volume 4.
101. Rothkopf, M.H. Scheduling Independent Tasks on Parallel Processors. *Manag. Sci.* **1966**, *12*, 437–447.
102. Sahni, S.K. Algorithms for scheduling independent tasks. *J. ACM (JACM)* **1976**, *23*, 116–127.
103. Karp, R.M. Dynamic programming meets the principle of inclusion and exclusion. *Oper. Res. Lett.* **1982**, *1*, 49–51.
104. O'Neil, T.E. Sub-exponential algorithms for 0/1 knapsack and bin packing, 2011.
105. Lenté, C.; Liedloff, M.; Soukhal, A.; T'Kindt, V. On an extension of the Sort & Search method with application to scheduling theory. *Theor. Comput. Sci.* **2013**, *511*, 13–22.
106. Jansen, K.; Land, F.; Land, K. Bounding the running time of algorithms for scheduling and packing problems. *SIAM J. Discret. Math.* **2016**, *30*, 343–366.
107. Chen, L.; Jansen, K.; Zhang, G. On the optimality of approximation schemes for the classical scheduling problem. In Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete algorithms, SIAM, Portland, OR, USA, 5–7 January 2014; pp. 657–668.
108. Goemans, M.X.; Rothvoß, T. Polynomiality for bin packing with a constant number of item types. *J. ACM (JACM)* **2020**, *67*, 1–21.
109. Chen, L.; Marx, D.; Ye, D.; Zhang, G. Parameterized and Approximation Results for Scheduling with a Low Rank Processing Time Matrix. In Proceedings of the 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017), Hannover, Germany, 8–11 March 2017; Leibniz International Proceedings in Informatics (LIPIcs); Vollmer, H., Vallée, B., Eds.; Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2017; Volume 66, pp. 22:1–22:14.
110. Fisher, M.L. Optimal Solution of Scheduling Problems Using Lagrange Multipliers: Part I. *Oper. Res.* **1973**, *21*, 1114–1127.
111. Fisher, M.L. Optimal Solution of Scheduling Problems Using Lagrange Multipliers: Part II. In Proceedings of the Symposium on the Theory of Scheduling and Its Applications, Nashville, TN, USA, 14 November 1973; Elmaghraby, S.E., Ed.; Springer: Berlin/Heidelberg, Germany, 1973; pp. 294–318.
112. Elmaghraby, S.E.; Elimam, A.A. Knapsack-Based Approaches to the Makespan Problem on Multiple Processors. *A I I E Trans.* **1980**, *12*, 87–96.
113. Mokotoff, E. An exact algorithm for the identical parallel machine scheduling problem. *Eur. J. Oper. Res.* **2004**, *152*, 758–769.
114. Dell'Amico, M.; Iori, M.; Martello, S.; Monaci, M. Heuristic and exact algorithms for the identical parallel machine scheduling problem. *INFORMS J. Comput.* **2008**, *20*, 333–344.
115. Korf, R.E. Multi-way number partitioning. In Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence, Pasadena, CA, USA, 14–16 July 2009; pp. 538–543.
116. Korf, R.E. A hybrid recursive multi-way number partitioning algorithm. In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, 16–22 July 2011.
117. Moffitt, M.D. Search strategies for optimal multi-way number partitioning. In Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, Beijing, China, 3–9 August 2013; pp. 623–629.
118. Korf, R.; Schreiber, E. Optimally Scheduling Small Numbers of Identical Parallel Machines. In Proceedings of the International Conference on Automated Planning and Scheduling, Rome, Italy, 10–14 June 2013; Volume 23, pp. 144–152.
119. Schreiber, E.L.; Korf, R.E. Improved bin completion for optimal bin packing and number partitioning. In Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, Beijing, China, 3–9 August 2013; pp. 651–658.
120. Korf, R.E.; Schreiber, E.L.; Moffitt, M.D. Optimal Sequential Multi-Way Number Partitioning. In Proceedings of the International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, FL, USA, 6–8 January 2014.
121. Schreiber, E.; Korf, R. Cached iterative weakening for optimal multi-way number partitioning. In Proceedings of the AAAI Conference on Artificial Intelligence, Québec City, QC, Canada, 27–31 July 2014; Volume 28.
122. KOWALCZYK, D. Branch-and-Price Algorithms for Scheduling Problems. Ph.D. Thesis, KU LEUVEN, Leuven, Belgium, 2018.
123. Mrad, M.; Souayah, N. An arc-flow model for the makespan minimization problem on identical parallel machines. *IEEE Access* **2018**, *6*, 5300–5307.
124. Gharbi, A.; Bamatraf, K. An Improved Arc Flow Model with Enhanced Bounds for Minimizing the Makespan in Identical Parallel Machine Scheduling. *Processes* **2022**, *10*, 2293.
125. Martello, S.; Toth, P. Lower bounds and reduction procedures for the bin packing problem. *Discret. Appl. Math.* **1990**, *28*, 59–70.
126. Belov, G.; Scheithauer, G. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *Eur. J. Oper. Res.* **2006**, *171*, 85–106.
127. Graham, R.L. Bounds for certain multiprocessing anomalies. *Bell Syst. Tech. J.* **1966**, *45*, 1563–1581.
128. Graham, R.L. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* **1969**, *17*, 416–429.

129. Coffman, E.G., Jr.; Garey, M.R.; Johnson, D.S. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM J. Comput.* **1978**, *7*, 1–17.
130. Mokotoff, E.; Jimeno, J.L.; Gutiérrez, A.I. List scheduling algorithms to minimize the makespan on identical parallel machines. *Trans. Oper. Res.* **2001**, *9*, 243–269.
131. Johnson, D.S. Fast allocation algorithms. In Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT 1972), Washington, DC, USA, 25–27 October 1972; IEEE: Piscataway, NJ, USA, 1972; pp. 144–154.
132. Friesen, D.K.; Langston, M.A. Evaluation of a MULTIFIT-based scheduling algorithm. *J. Algorithms* **1986**, *7*, 35–59.
133. Lee, C.Y.; Massey, J. Multiprocessor scheduling: An extension of the MULTIFIT algorithm. *J. Manuf. Syst.* **1988**, *7*, 25–32.
134. Lee, C.Y.; Massey, J. Multiprocessor scheduling: combining LPT and MULTIFIT. *Discret. Appl. Math.* **1988**, *20*, 233–242.
135. França, P.M.; Gendreau, M.; Laporte, G.; Müller, F.M. A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective. *Comput. Oper. Res.* **1994**, *21*, 205–210.
136. Riera, J.; Alcaide, D.; Sicilia, J. Approximate algorithms for the P|| C max problem. *Trans. Oper. Res.* **1996**, *4*, 345–359.
137. Gupta, J.N.D.; Ruiz-Torres, A.J. A LISTFIT heuristic for minimizing makespan on identical parallel machines. *Prod. Plan. Control* **2001**, *12*, 28–36.
138. Jimeno, J.L.; Mokotoff, E.; Pérez, J. A Constructive Algorithm to Minimise the Makespan on Identical Parallel Machines. In Proceedings of the Eighth International Workshop on Project Management and Scheduling, Valencia, Spain, 3–5 April 2002;
139. Alvim, A.C.; Ribeiro, C.C. A hybrid bin-packing heuristic to multiprocessor scheduling. In Proceedings of the International Workshop on Experimental and Efficient Algorithms, Angra dos Reis, Brazil, 25–28 May 2004; Springer: Berlin/Heidelberg, Germany, 2004; pp. 1–13.
140. Paletta, G.; Pietramala, P. A new approximation algorithm for the nonpreemptive scheduling of independent jobs on identical parallel processors. *SIAM J. Discret. Math.* **2007**, *21*, 313–328.
141. Gualtieri, M.I.; Paletta, G.; Pietramala, P. A new n log n algorithm for the identical parallel machine scheduling problem. *Int. J. Contemp. Math. Sciences* **2008**, *3*, 25–36.
142. Gualtieri, M.; Pietramala, P.; Rossi, F. Heuristic Algorithms for Scheduling Jobs on Identical Parallel Machines via Measures of Spread. *IAENG Int. J. Appl. Math.* **2009**, *39*.
143. Chiaselotti, G.; Gualtieri, M.I.; Pietramala, P. Minimizing the makespan in nonpreemptive parallel machine scheduling problem. *J. Math. Model. Algorithms* **2010**, *9*, 39–51.
144. Kuruvilla, A.; Paletta, G. Minimizing makespan on identical parallel machines. *Int. J. Oper. Res. Inf. Syst. (IJORIS)* **2015**, *6*, 19–29.
145. Paletta, G.; Ruiz-Torres, A.J. Partial solutions and multifit algorithm for multiprocessor scheduling. *J. Math. Model. Algorithms Oper. Res.* **2015**, *14*, 125–143.
146. Della Croce, F.; Scatamacchia, R. The longest processing time rule for identical parallel machines revisited. *J. Sched.* **2020**, *23*, 163–176.
147. Davidović, T. Exhaustive List–Scheduling heuristic for dense task graphs. *YUJOR* **2000**, *10*, 123–136.
148. Öztürk, S. A New Heuristic Algorithm for Minimizing the Makespan on Identical Parallel Machines. Ph.D. Thesis, Marmara Universitesi, Istanbul, Turkey, 2008.
149. Johnson, D.S.; Demers, A.; Ullman, J.D.; Garey, M.R.; Graham, R.L. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.* **1974**, *3*, 299–325.
150. Martello, S.; Toth, P. Worst-case analysis of greedy algorithms for the subset-sum problem. *Math. Program.* **1984**, *28*, 198–205.
151. Martello, S.; Toth, P. *Knapsack Problems: Algorithms and Computer Implementations*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 1990.
152. Pisinger. Dynamic Programming on the Word RAM. *Algorithmica* **2003**, *35*, 128–145.
153. Nichols, R.; Bulfin, R.; Parker, R. An interactive procedure for minimizing makespan on parallel processors. *Int. J. Prod. Res.* **1978**, *16*.
154. Finn, G.; Horowitz, E. A linear time approximation algorithm for multiprocessor scheduling. *BIT Numer. Math.* **1979**, *19*, 312–320.
155. Langston, M.A. Improved 0/1-interchange scheduling. *BIT Numer. Math.* **1982**, *22*, 282–290.
156. Barr, R.S.; Ross, G.T. *A Linked List Data Structure for a Binary Knapsack Algorithm*; Research Report CC 232; Center for Cybernetic Studies, University of Texas, Austin, USA, 1 July 1975.
157. Blackstone, J.H., Jr.; Phillips, D.T. An improved heuristic for minimizing makespan among m identical parallel processors. *Comput. Ind. Eng.* **1981**, *5*, 279–287.
158. Langston, M.A. Remarks on the makespan minimization problem. *Comput. Ind. Eng.* **1984**, *8*, 193–195.
159. Ho, J.C.; Wong, J.S. Makespan minimization for m parallel identical processors. *Nav. Res. Logist. (NRL)* **1995**, *42*, 935–948.
160. Ghomi, S.M.T.F.; Ghazvini, F.J. A pairwise interchange algorithm for parallel machine scheduling. *Prod. Plan. Control* **1998**, *9*, 685–689.
161. Costa, A.M.; Vargas, P.A.; Von Zuben, F.J.; Franca, P.M. Makespan minimization on parallel processors: an immune-based approach. In Proceedings of the 2002 Congress on Evolutionary Computation, CEC'02 (Cat. No. 02TH8600), Honolulu, HI, USA, 12–17 May 2002; IEEE: Piscataway, NJ, USA, 2002; Volume 1, pp. 920–925.

162. Frangioni, A.; Necciari, E.; Scutellà, M.G. *A Multi-Exchange Neighborhood for Minimum Makespan Machine*; Technical Report; Università di Pisa: Pisa, Italy, 2000.
163. Frangioni, A.; Necciari, E.; Scutella, M.G. A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems. *J. Comb. Optim.* **2004**, *8*, 195–220.
164. Paletta, G.; Vocaturo, F. A composite algorithm for multiprocessor scheduling. *J. Heuristics* **2011**, *17*, 281–301.
165. Habiba, H.; Hassam, A.; Sari, Z.; Amine, C.M.; Souad, T. Minimizing Makespan on Identical Parallel Machines. In Proceedings of the 2019 International Conference on Applied Automation and Industrial Diagnostics (ICAAID), Elazig, Turkey, 25–27 September 2019; Volume 1, pp. 1–6.
166. Ostojić, D.; Davidović, T.; Jakšić-Krüger, T.; Ramljak, D. Comparative Analysis of Heuristic Approaches to P||Cmax. In Proceedings of the 11th International Conference on Operations Research and Enterprise Systems, ICORES, Virtual, 3–5 February 2022; SciTePress: 2022; pp. 259–266.
167. Vazirani, V.V. *Approximation Algorithms*; Springer: Berlin/Heidelberg, Germany, 2003.
168. Hochbaum, D.S.; Shmoys, D.B. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. In Proceedings of the 26th Annual Symposium on Foundations of Computer Science (sfcs 1985), Portland, OR, USA, 21–23 October 1985; IEEE: Piscataway, NJ, USA, 1985; pp. 79–89.
169. Leung, J.Y. Bin packing with restricted piece sizes. *Inf. Process. Lett.* **1989**, *31*, 145–149.
170. Hochbaum, D.S. Approximation algorithms for NP-hard problems. *ACM Sigact News* **1997**, *28*, 40–52.
171. Jansen, K. An EPTAS for scheduling jobs on uniform processors: Using an MILP relaxation with a constant number of integral variables. *SIAM J. Discret. Math.* **2010**, *24*, 457–485.
172. Jansen, K.; Rohwedder, L. On integer programming and convolution. In Proceedings of the 10th Innovations in Theoretical Computer Science Conference (ITCS 2019), San Diego, CA, USA, 10–12 January 2019; Schloss-Dagstuhl-Leibniz Zentrum für Informatik: 2019.
173. Jansen, K.; Klein, K.M.; Verschae, J. Closing the gap for makespan scheduling via sparsification techniques. *Math. Oper. Res.* **2020**, *45*, 1371–1392.
174. Berndt, S.; Deppert, M.A.; Jansen, K.; Rohwedder, L. Load balancing: The long road from theory to practice. In Proceedings of the 2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX), SIAM, Alexandria, VA, USA, 9–10 January 2022; pp. 104–116.
175. Jansen, K.; Rohwedder, L. On integer programming, discrepancy, and convolution. *Math. Oper. Res.* **2023**, *48*, 1481–1495.
176. Sreenivas, P.; Saheb, S.K.P.; Yohan, M. An overview of harmony search algorithm applied in identical parallel machine scheduling. *Recent Trends Mech. Eng. Sel. Proc. ICIME 2019* **2020**, pp. 709–714.
177. Davidović, T.; Šelmić, M.; Teodorović, D.; Ramljak, D. Bee colony optimization for scheduling independent tasks to identical processors. *J. Heuristics* **2012**, *18*, 549–569.
178. Hübscher, R.; Glover, F. Applying tabu search with influential diversification to multiprocessor scheduling. *Comput. Oper. Res.* **1994**, *21*, 877–884.
179. Brucker, P.; Hurink, J.; Werner, F. Improving local search heuristics for some scheduling problems. Part II. *Discret. Appl. Math.* **1997**, *72*, 47–69. Models and Algorithms for Planning and Scheduling Problems.
180. Thesen, A. Design and evaluation of tabu search algorithms for multiprocessor scheduling. *J. Heuristics* **1998**, *4*, 141–160.
181. Min, L.; Cheng, W. A genetic algorithm for minimizing the makespan in the case of scheduling identical parallel machines. *Artif. Intell. Eng.* **1999**, *13*, 399–403.
182. Lee, W.C.; Wu, C.C.; Chen, P. A simulated annealing approach to makespan minimization on identical parallel machines. *Int. J. Adv. Manuf. Technol.* **2006**, *31*, 328–334.
183. Tang, L.; Luo, J. A new ILS algorithm for parallel machine scheduling problems. *J. Intell. Manuf.* **2006**, *17*, 609–619.
184. Akyol, D.E.; Bayhan, G.M. Minimizing makespan on identical parallel machines using neural networks. In Proceedings of the International Conference on Neural Information Processing, Hong Kong, China, 3–6 October 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 553–562.
185. Akyol, D.E. Identical parallel machine scheduling with dynamical networks using time-varying penalty parameters. In *Multiprocessor Scheduling: Theory Applications*; I-TECH Education and Publishing: Vienna, Austria, 2007; pp. 293–314.
186. Yu, A.; Gu, X. An Improved Transiently Chaotic Neural Network Approach for Identical Parallel Machine Scheduling. In *Proceedings of the Advances in Cognitive Neurodynamics ICCN 2007*; Wang, R., Shen, E., Gu, F., Eds.; Springer: Amsterdam, The Netherlands, 2008; pp. 909–913.
187. Davidović, T.; Janićijević, S. Heuristic Approach to Scheduling Independent Tasks on Identical Processor. In Proceedings of the Symposium on Information Technology, YUINFO, Moscow, Russia, 23–25 January 2009; pp. 1–6.
188. Davidović, T.; Janićijević, S. VNS for Scheduling Independent Tasks on Identical Processors. In Proceedings of the 36th Symposium on Operational Research, SYMOPIS, Scottsdale, AZ, USA, 22–25 September 2009; pp. 301–304.

189. Kashan, A.H.; Karimi, B. A discrete particle swarm optimization algorithm for scheduling parallel machines. *Comput. Ind. Eng.* **2009**, *56*, 216–223.
190. Chen, J.; Pan, Q.K.; Li, H. Harmony search algorithm with dynamic subpopulations for scheduling identical parallel machines. In Proceedings of the 2010 Sixth International Conference on Natural Computation, Yantai, China, 10–12 August 2010; IEEE: Piscataway, NJ, USA, 2010; Volume 5, pp. 2369–2373.
191. Chen, J.; Liu, G.L.; Lu, R. Discrete harmony search algorithm for identical parallel machine scheduling problem. In Proceedings of the 30th Chinese Control Conference, Yantai, China, 22–24 July 2011; IEEE: Piscataway, NJ, USA, 2011; pp. 5457–5461.
192. Jing, C.; Jun-qing, L. Efficient variable neighborhood search for identical parallel machines scheduling. In Proceedings of the 31st Chinese Control Conference, Hefei, China, 25–27 July 2012; pp. 7228–7232.
193. Laha, D. A simulated annealing heuristic for minimizing makespan in parallel machine scheduling. In Proceedings of the Swarm, Evolutionary, and Memetic Computing: Third International Conference, SEMCCO 2012, Proceedings 3, Bhubaneswar, India, 20–22 December 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 198–205.
194. Sevkli, M.; Sevkli, A.Z. A stochastically perturbed particle swarm optimization for identical parallel machine scheduling problems. In *Bio-Inspired Computational Algorithms and Their Applications*; 2012; pp. 371–382.
195. Chen, J.; Pan, Q.K.; Wang, L.; Li, J.Q. A hybrid dynamic harmony search algorithm for identical parallel machines scheduling. *Eng. Optim.* **2012**, *44*, 209–224.
196. Laha, D.; Behera, D.K. An Improved Cuckoo Search Algorithm for Parallel Machine Scheduling. In Proceedings of the Swarm, Swarm, Evolutionary, and Memetic Computing: 5th International Conference, SEMCCO 2014, Bhubaneswar, India, December 18-20, 2014, Revised Selected Papers 5; Springer International Publishing: Berlin/Heidelberg, Germany, 2015; pp. 788–800.
197. Kashan, A.H.; Keshmiry, M.; Dahooie, J.H.; Abbasi-Pooya, A. A simple yet effective grouping evolutionary strategy (GES) algorithm for scheduling parallel machines. *Neural Comput. Appl.* **2018**, *30*, 1925–1938.
198. Laha, D.; Gupta, J.N. An improved cuckoo search algorithm for scheduling jobs on identical parallel machines. *Comput. Ind. Eng.* **2018**, *126*, 348–360.
199. Alharkan, I.; Bamatraf, K.; Noman, M.A.; Kaid, H.; Abouel Nasr, E.S.; El-Tamimi, A.M. An order effect of neighborhood structures in variable neighborhood search algorithm for minimizing the makespan in an identical parallel machine scheduling. *Math. Probl. Eng.* **2018**, *2018*, 3586731.
200. Kamaraj, S.; Saravanan, M. Optimisation of identical parallel machine scheduling problem. *Int. J. Rapid Manuf.* **2019**, *8*, 123–132.
201. Ghalami, L.; Grosu, D. Scheduling parallel identical machines to minimize makespan: A parallel approximation algorithm. *J. Parallel Distrib. Comput.* **2019**, *133*, 221–231.
202. Davidović, T.; Jakšić-Krüger, T.; Ramljak, D.; Šelmić, M.; Teodorović, D. Parallelization strategies for bee colony optimization based on message passing communication protocol. *Optimization* **2013**, *62*, 1113–1142.
203. Abdelsalam, K.M.; Khamis, S.M.; Bahig, H.M.; Bahig, H.M. A multicore-based algorithm for optimal multi-way number partitioning. *Int. J. Inf. Technol.* **2023**, *15*, 2929–2940.
204. Kedia, S. *A Job Scheduling Problem with Parallel Processors*; Unpublished Report; Department of Industrial and Operations Engineering, University of Michigan: Ann Arbor, MI, USA, 1971.
205. Ostojić, D.; Jolović, M.; Drašković, R.; Fellague, A. Efficient Generation of Diverse Instances for P||Cmax Solver Evaluation. In Proceedings of the First Deep Tech Open Science Day Conference, Kragujevac , Serbia, 5 April 2024.
206. Ostojić, D.; Zarges, C.; Davidović, T.; Ramljak, D. Polynomial Regression Model for Standardizing Large Precision Instances for P||Cmax problem. In Proceedings of the Book of Abstracts of the Artificial Intelligence Conference, San Francisco, CA, USA, 10–13 June 2024.
207. Dell'Amico, M.; Martello, S. A note on exact algorithms for the identical parallel machine scheduling problem. *Eur. J. Oper. Res.* **2005**, *160*, 576–578.
208. Falkenauer, E. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics* **1996**, *2*, 5–30.
209. Davidović, T.; Crainic, T.G. Benchmark-Problem Instances for Static Scheduling of Task Graphs with Communication Delays on Homogeneous Multiprocessor Systems. *Comput. Oper. Res.* **2006**, *33*, 2155–2177.
210. Tobita, T.; Kasahara, H. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *J. Sched.* **2002**, *5*, 379–394.
211. Dell'Amico, M.; Iori, M.; Martello, S.; Monaci, M. A note on exact and heuristic algorithms for the identical parallel machine scheduling problem. *J. Heuristics* **2012**, *18*, 939–942.
212. Beiranvand, V.; Hare, W.; Lucet, Y. Best practices for comparing optimization algorithms. *Optim. Eng.* **2017**, *18*, 815–848.
213. Bartz-Beielstein, T.; Doerr, C.; Berg, D.v.d.; Bossek, J.; Chandrasekaran, S.; Eftimov, T.; Fischbach, A.; Kerschke, P.; La Cava, W.; Lopez-Ibanez, M.; et al. Benchmarking in optimization: Best practice and open issues. *arXiv preprint* **2020**, arXiv:2007.03488.
214. Hooker, J.N. Needed: An empirical science of algorithms. *Oper. Res.* **1994**, *42*, 201–212.
215. Rardin, R.L.; Uzsoy, R. Experimental evaluation of heuristic optimization algorithms: A tutorial. *J. Heuristics* **2001**, *7*, 261–304.

216. Barr, R.S.; Golden, B.L.; Kelly, J.P.; Resende, M.G.; Stewart, W.R. Designing and reporting on computational experiments with heuristic methods. *J. Heuristics* **1995**, *1*, 9–32.
217. Krüger, T.J. Development, Implementation and Theoretical Analysis of the Bee Colony Optimization Meta-Heuristic Method. Ph.D. Thesis, University of Novi Sad, Novi Sad, Serbia, 2017.
218. Biedrzycki, R. Comparison with State-of-the-Art: Traps and Pitfalls. In Proceedings of the 2021 IEEE Congress on Evolutionary Computation (CEC), Krakow, Poland, 1 July 2021; IEEE: Pisccataway, NJ, USA, 2021; pp. 863–870.
219. Behera, D.K.; Laha, D. Comparison of heuristics for identical parallel machine scheduling. *Adv. Mater. Res.* **2012**, *488*, 1708–1712.
220. Alvim, A.C.; Ribeiro, C. A hybrid heuristic for bin-packing and multiprocessor scheduling. In Proceedings of the International Workshop on Experimental and Efficient Algorithms, Angra dos Reis, Brazil, 25–28 May 2004.
221. Alba, E.; Luque, G.; Nesmachnow, S. Parallel metaheuristics: recent advances and new trends. *Int. Trans. Oper. Res.* **2013**, *20*, 1–48.
222. Crainic, T. Parallel metaheuristics and cooperative search. In *Handbook of Metaheuristics*; 2019; pp. 419–451.
223. Xiao, X. A direct proof of the 4/3 bound of lpt scheduling rule. In Proceedings of the 2017 5th International Conference on Frontiers of Manufacturing Science and Measuring Technology (FMSMT 2017). Atlantis Press, Taiyuan, China, 24–25 June 2017; pp. 486–489.
224. Friesen, D.K. Tighter bounds for the multifit processor scheduling algorithm. *SIAM J. Comput.* **1984**, *13*, 170–181.
225. Yue, M. On the exact upper bound for the multifit processor scheduling algorithm. *ANnals Oper. Res.* **1990**, *24*, 233–259.
226. Fischetti, M.; Martello, S. Worst-case analysis of the differencing method for the partition problem. *Math. Program.* **1987**, *37*, 117–120.
227. Michiels, W.; Korst, J.; Aarts, E.; Leeuwen, J. Performance ratios of the Karmarkar–Karp differencing method. *J. Comb. Optim.* **2007**, *13*, 19–32.
228. Michiels, W. Performance Ratios for the Differencing Method. Ph.D. Thesis, Eindhoven University of Technology (TU/e), Frits Philips Inst. Quality Management, Groene Loper 3, 5612 AE Eindhoven, The Netherlands, 2004. https://doi.org/10.6100/IR573763.
229. Boettcher, S.; Mertens, S. Analysis of the Karmarkar-Karp differencing algorithm. *Eur. Phys. J. B* **2008**, *65*, 131–140.
230. Coffman, E.G., Jr.; Whitt, W. Recent Asymptotic Results in the Probabilistic Analysis of Schedule Makespans. In *Scheduling Theory and Its Applications*; Chrètienne, P., Coffman, E.G., Jr., Lenstra, J.K., Liu, Z., Eds.; Wiley: New York, NY, USA, 1997; pp. 15–31.
231. Yakir, B. The Differencing Algorithm LDM for Partitioning: A Proof of a Conjecture of Karmarkar and Karp. *Math. Oper. Res.* **1996**, *21*, 85–99.
232. Ho, J.C.; Massabò, I.; Paletta, G.; Ruiz-Torres, A.J. A note on posterior tight worst-case bounds for longest processing time schedules. *4OR* **2019**, *17*, 97–107.
233. Blocher, J.D.; Sevastyanov, S. A note on the Coffman–Sethi bound for LPT scheduling. *J. Sched.* **2015**, *18*, 325–327.
234. Chen, B. A note on LPT scheduling. *Oper. Res. Lett.* **1993**, *14*, 139–142.
235. Blocher, J.D.; Chand, S. Scheduling of parallel processors: A posterior bound on LPT sequencing and a two-step algorithm. *Nav. Res. Logist. (NRL)* **1991**, *38*, 273–287.
236. Dobson, G. Scheduling Independent Tasks on Uniform Processors. *SIAM J. Comput.* **1984**, *13*, 705–716.
237. Coffman, E.G., Jr.; Sethi, R. A Generalized Bound on LPT Sequencing. In Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation, SIGMETRICS '76, New York, NY, USA, 29–31 March 1976; p. 306–310.
238. Paletta, G.; Vocaturo, F. A short note on an advance in estimating the worst-case performance ratio of the MPS algorithm. *SIAM J. Discret. Math.* **2010**, *23*, 2198–2203.
239. Benoit, A.; Canon, L.C.; Elghazi, R.; Héam, P.C. Asymptotic Performance and Energy Consumption of SLACK. In Proceedings of the Euro-Par 2023: Parallel Processing, Limassol, Cyprus, 28 August–1 September 2023; Cano, J., Dikaiakos, M.D., Papadopoulos, G.A., Pericàs, M., Sakellariou, R., Eds.; Springer Nature: Cham, Switzerland, 2023.
240. Lee, M.; Lee, K.; Pinedo, M. Tight approximation bounds for the LPT rule applied to identical parallel machines with small jobs. *J. Sched.* **2022**, *25*, 721–740.
241. Benoit, A.; Canon, L.C.; Elghazi, R.; Héam, P.C. Update on the Asymptotic Optimality of LPT. In Proceedings of the Euro-Par 2021: Parallel Processing, Lisbon, Portugal, 1–3 September 2021; Sousa, L., Roma, N., Tomás, P., Eds.; Springer Nature: Cham, Switzerland, 2021; pp. 55–69.
242. Frenk, J.B.G.; Rinnooy Kan, A.H.G. The Asymptotic Optimality of the LPT Rule. *Math. Oper. Res.* **1987**, *12*, 241–254.
243. Loulou, R. Tight Bounds and Probabilistic Analysis of Two Heuristics for Parallel Processor Scheduling. *Math. Oper. Res.* **1984**, *9*, 142–150.
244. Coffman, E.G., Jr.; Frederickson, G.N.; Lueker, G.S. Probabilistic Analysis of the LPT Processor Scheduling Heuristic. In *Deterministic and Stochastic Scheduling: Proceedings of the NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems held in Durham, England, 6–17 July 1981*; Dempster, M.A.H., Lenstra, J.K., Rinnooy Kan, A.H.G., Eds.; Springer: Amsterdam, The Netherlands, 1982; pp. 319–331.
245. Kao, T.Y.; Elsayed, E.A. Performance of the LPT algorithm in multiprocessor scheduling. *Comput. Oper. Res.* **1990**, *17*, 365–373.

246. Coffman, E.G., Jr.; Lueker, G.S.; Rinnooy Kan, A.H.G. Asymptotic Methods in the Probabilistic Analysis of Sequencing and Packing Heuristics. *Manag. Sci.* **1988**, *34*, 266–290.
247. Bruno, J.L.; Downey, P.J. Probabilistic Bounds on the Performance of List Scheduling. *SIAM J. Comput.* **1986**, *15*, 409–417.
248. Coffman, E.G., Jr.; Gilbert, E.N. On the Expected Relative Performance of List Scheduling. *Oper. Res.* **1985**, *33*, 548–561.
249. Han, S.; Hong, D.; Leung, J.Y.T. On the Asymptotic Optimality of Multiprocessor Scheduling Heuristics for the Makespan Minimization Problem. *ORSA J. Comput.* **1995**, *7*, 201–204.
250. Coffman, E.G., Jr.; Langston, M.A. A performance guarantee for the greedy set-partitioning algorithm. *Acta Inform.* **1984**, *21*, 409–415.
251. Alvim, A.C.; Ribeiro, C.C. A Hybrid Bin-Packing Heuristic to Multiprocessor Scheduling: Detailed Computational Results. In Proceedings of the International Workshop on Experimental and Efficient Algorithms, Angra dos Reis, Brazil, 25–28 May 2004.
252. Berretta, R.; Moscato, P., The number partitioning problem: an open challenge for evolutionary computation? In *New Ideas in Optimization*; McGraw-Hill Ltd.: London, UK, 1999; p. 261–278.
253. Pan, Q.K.; Tasgetiren, M.F.; Liang, Y.C. A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem. *Comput. Oper. Res.* **2008**, *35*, 2807–2839.
254. Tasgetiren, M.F.; Liang, Y.C.; Sevkli, M.; Gencyilmaz, G. A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. *Eur. J. Oper. Res.* **2007**, *177*, 1930–1947.
255. Fernandez, A.; Insfran, E.; Abrahão, S. Usability evaluation methods for the web: A systematic mapping study. *Inf. Softw. Technol.* **2011**, *53*, 789–817.
256. Filippi, C.; Romanin-Jacur, G. Exact and approximate algorithms for high-multiplicity parallel machine scheduling. *J. Sched.* **2009**, *12*, 529–541.
257. Filippi, C. An approximate algorithm for a high-multiplicity parallel machine scheduling problem. *Oper. Res. Lett.* **2010**, *38*, 312–317.
258. Gabay, M. High-Multiplicity Scheduling and Packing Problems: Theory and Applications. Ph.D. Thesis, Grenoble, France, 2014.
259. Oosterwijk, T. Approximation Algorithms in Allocation, Scheduling and Pricing. Ph.D. Thesis, Universitaire Pers Maastricht, Maastricht, The Netherlands, 2018.
260. Hoos, H.H.; Stützle, T. Empirical analysis of randomized algorithms. In *Handbook of Approximation Algorithms and Metaheuristics*; 2018.