Solving Variants of Shop Scheduling Problems with Deep Learning Methodologies

Andrea Corsini*

XXXVI Cycle of the Doctoral School in
Industrial Innovation Engineering

University of Modena and Reggio Emilia

March 1^{st}, 2024

Supervisor: Professor Mauro Dell'Amico.
School Coordinator: Professor Franco Zambonelli.

Keywords: Operations Research; Scheduling; Job Shop; Deep Learning; Heuristics.

*andrea.corsini@unimore.it

Abstract

In the landscape of modern industries, scheduling problems are pivotal as they influence an organization's competitiveness, operational costs, and capacity to meet demands. This work specifically delves into variants of shop scheduling problems involving the planning of production or logistics processes with precedence among tasks/jobs. Our attention is mainly directed toward two distinct types of shop scheduling problems: the classical Job Shop Scheduling and a generalization of the Flexible Flow Shop Scheduling. The former, extensively studied in the literature, stands as a perfect candidate for developing innovative resolution methodologies. Conversely, the latter stems from a real-life problem that is first introduced in this work. Our research contributes to the existing literature in two different directions. Motivated by the recent success of learning methodologies in other fields, the first part of this work focuses on deep learning applications to shop scheduling problems. When strategically employed, these data-driven methodologies can complement and enhance existing methods, offering additional flexibility and customization. Although several recent works addressing Flow and Job Shop variants exist, mostly relying on Reinforcement Learning, they still lag behind classic methodologies. Therefore, in one research direction, we investigate and advance the applicability of supervised learning paradigms for the resolution of a renowned problem, namely Job Shop Scheduling. Specifically, we employ supervised strategies to *enhance a Tabu Search* meta-heuristic by guiding its exploration with a sequential deep learning model. In addition, we devise an end-to-end learning system that relies on a *self-supervised training strategy*, neither requiring expensive optimal solutions nor resorting to Reinforcement Learning. Notably, this latter method outperforms all the recent deep learning approaches for the Job Shop. In the second part of this work, we study novel and intricate scheduling problems. One of these problems arises from a manufacturing industry, where *parallel machines and workforce constraints intertwine with precedence ones*. Although these constraints are separately studied in variants of scheduling problems, their confluence creates a distinctive landscape that challenges and modifies assumptions upon which traditional methodologies rely. Thus, we formalize the problem, develop lower bounds to evaluate algorithms, adapt baseline heuristics from related scheduling variants, and propose a new heuristic method to efficiently tackle this problem. In a different context, we tackle variants of a recently presented problem, requiring the *routing of vehicles and the scheduling of drones for parcel deliveries*. We propose new Constraint Programming models and introduce valid inequalities for improving exact methods. These proposals are extensively benchmarked and compared against proposals in the literature to demonstrate their effectiveness. We believe that our work makes a step forward in enriching the repertoire of problem-solving methodologies and contributes to adapting techniques to the complexity of real-world industrial challenges.

Contents

1 Introduction **5**

2 Preliminaries **9**
2.1 Machine & Deep Learning 9
2.1.1 Reinforcement Learning 10
2.1.2 Semi- and Self-Supervised Learning 12
2.1.3 The Pointer Neural Network 13
2.2 Job Shop Scheduling Problem 15
2.2.1 Disjunctive Graph 15
2.2.2 Constructing Solutions 16
2.2.3 Improving Solutions: Meta-heuristics 17
2.2.4 Machine Learning Applications 19

3 Enhancing Tabu Search for the Job Shop Problem **22**
3.1 Learning the Quality of Machine Permutations 23
3.2 The Neural Network Oracle 26
3.3 Tabu Search 27
3.4 The Supervised Dataset 28
3.5 Results 30
3.5.1 Oracle performance 30
3.5.2 Tabu Search performance 34
3.5.3 Comparison with Reinforcement Learning 35
3.6 Final Remarks 36

4 Self-Labeling the Job Shop Problem **38**
4.1 Proposed Pointer Neural Network 39
4.2 Self-Labeling Training Strategy 42
4.3 Experimental Setup 44
4.4 Results 46
4.4.1 Performance on Benchmarks 46
4.4.2 Execution Times 48
4.4.3 The effect of β on Training 49
4.4.4 The effect of β on Testing 50
4.5 Final Remarks 51

5 Solving a Complex Shop Problem **52**
5.1 The manufacturing problem 53
5.2 Related problems 55
5.3 Problem formulation 56
5.3.1 Mathematical model 57
5.3.2 Lower bounds 58
5.4 Heuristics 59

5.4.1 Priority Dispatching Rules . . . 59
5.4.2 Job Sequencing . . . 60
5.4.3 A novel PDR Approach . . . 62
5.5 Experimental setup . . . 63
5.5.1 Benchmarks . . . 64
5.6 Results . . . 65
5.6.1 Typed Instances . . . 65
5.6.2 Performance for Large Computing Times . . . 67
5.6.3 Performance Under Different Shop Floor Configurations . 67
5.6.4 Real-life Instances . . . 69
5.7 Final Remarks . . . 70

6 Parallel Drone Scheduling Vehicle Routing Problems with Collective Drones 71
6.1 Parallel Drone Scheduling Traveling Salesman Problem with Collective Drones . . . 74
6.1.1 Problem Description . . . 74
6.1.2 A Constraint Programming model . . . 74
6.1.3 Valid inequality . . . 77
6.2 Parallel Drone Scheduling Vehicle Routing Problem with Collective Drones . . . 78
6.2.1 Problem Description . . . 78
6.2.2 A 2-indices Constraint Programming model . . . 78
6.2.3 A 3-indices Constraint Programming model . . . 80
6.2.4 A 3-indices Mixed Integer Linear Programming model . . 82
6.3 Results . . . 83
6.3.1 Benchmark Instances . . . 83
6.3.2 PDSTSP-c . . . 84
6.3.3 PDSVRP-c . . . 88
6.4 Final Remarks . . . 93

7 Conclusions 94

1 Introduction

The scheduling of tasks with finish-to-start precedences is a common problem across various industries, prompting extensive research by the Operations Research community. This has given rise to a large spectrum of methodologies, including (meta-)heuristics and exact methods, designed to effectively tackle renowned scheduling problems. Despite their advancements, these methodologies still exhibit inherent limitations in certain circumstances and often require cumbersome adaptation to address emerging problem variants.

Therefore, this work unfolds in two distinct parts. In the first part, we address some limitations of existing methodologies and introduce novel approaches by harnessing the power of Machine Learning (ML) techniques, which enabled several breakthroughs across different domains. Whereas, in the second part, we delve into the exploration of novel problems involving scheduling and propose effective resolution methodologies, without relying on learning techniques. This dual approach aims to advance the understanding and resolution of scheduling problems from traditional and new data-driven perspectives.

Within the realm of scheduling problems characterized by finish-to-start precedences, the Job Shop Scheduling (JSP) stands out as a classic optimization problem with many practical applications in manufacturing, production scheduling, and logistics [1, 2]. At its core, the JSP involves scheduling a set of jobs onto a set of machines, where each job has to be processed on the machines following a specific order. Although the JSP may target various optimization objectives (see [3] for examples), it frequently requires the minimization of the *makespan* – the overall time required to complete all jobs. In the first part of this work, we use the well-established Job Shop Scheduling Problem with the makespan objective as a case study.

Over the years, researchers have developed several methodologies for tackling the JSP. A common one is adopting exact methods such as Mixed Integer Linear Programming (MILP). While effective for small instances, standard MILP models often fall short in delivering optimal or high-quality solutions for medium and large-scale problems [4]. As a remedy, meta-heuristics have emerged as a widely embraced and effective alternative, capable of providing quality solutions within reasonable timeframes – usually minutes. Extensively studied and refined over

the years [5, 6, 7, 8, 9], these methods strike a delicate balance between solution quality and computational efficiency. Achieving the right balance while designing meta-heuristics requires both algorithmic expertise and a deep understanding of the problem at hand, especially when additional personalization has to be introduced. Under tighter time constraints and in applied variants of the JSP, like dynamic and rescheduling ones [10], Priority Dispatching Rules (PDR) are often preferred or adopted as baseline algorithms. The main issue of PDRs is their tendency to perform well in some cases and poorly in others, primarily due to their blind prioritization schema based on rigid hardcoded and hand-crafted rules [11].

For these reasons, recent works have started investigating ML techniques to partially alleviate some of these issues or create new and better neural solvers. After initial promising works, Reinforcement Learning (RL), one of the three main ML paradigms, demonstrated to be a valid paradigm for either improving [12] or creating resolution methodologies for the JSP [13, 14]. Despite the potential demonstrated by RL, training RL agents is in general a complex optimization task [15] and has reproducibility issues [16]. Therefore, we focus on supervised methodologies which have been less investigated and are subject to fewer of these issues.

In one research direction, we introduce a novel supervised learning task for enhancing and automating the creation of effective algorithms like meta-heuristics and decomposition heuristics. We arrived at the formulation of this task by addressing two fundamental questions: (i) what kind of information could enhance methodologies for solving the JSP? (ii) how can this information be learned in a supervised manner? These questions arise from the fact that not all the solutions to a JSP instance are feasible, i.e., respect the problem constraints, and for those feasible, the objective value (e.g., the makespan or the total tardiness) is not trivially derivable. Consequently, applying supervised learning to the JSP requires a learning task closely related to the objective function that may fit in the back-propagation algorithm.

We thus propose as a novel supervised learning task to *learn the quality of machine permutations*. Since having a method to judge machines is important, either for speeding up meta-heuristics or even in machine-based decomposition, we present an original methodology to learn the quality of machines using *sequential deep learning* and a *MILP solver*. Unlike existing approaches in the literature, such as the shifting bottleneck [17], we define the quality of a machine permutation as the *likelihood of finding this permutation in an optimal or near-optimal solution*.

In another research direction, we focus on designing an end-to-end supervised learning system that learns to construct high-quality solutions with a rather simple strategy, effectively outperforming most of the traditional constructive and RL approaches for the JSP. As mentioned above, supervised learning is generally not affected by training instability and reproducibility issues, but heavily relies on expensive annotations [18]. This is particularly problematic in combinatorial problems, where annotations in the form of optimal solutions are generally

produced with expensive exact methods.

To contrast labeling and still train in a supervised manner, we demonstrate how *Semi- and Self-Supervised learning* [18, 19] can be combined to train effective neural constructive algorithms. To the best of our knowledge, this is the first application of these paradigms to the JSP and any combinatorial problem [20].

In the second part of this work, we focus on studying and designing resolution methodologies for new complex problems still involving the scheduling of tasks. Specifically, we present a new problem that can be seen as a generalization of the *Flexible Flow Shop Problem*, encompassing characteristics of other scheduling problems, and a new mixed problem involving the routing of vehicles and the scheduling of drones.

Stemming from a real-life industrial setting, we delve into a scheduling problem encountered in off-road vehicle manufacturing – a sector pivotal to modern and automated industries. Off-road vehicles have a wide range of applications thanks to unique and distinctive characteristics, and in general, their manufacturing process is rather complex and challenging, involving parallel machines, special precedence structures, and reusable resources.

More in detail, the problem requires the manufacturing of vehicles according to a three-level assembly process. At the first level, the vehicle is equipped with all the necessary components (*e.g.*, wheels or engine), which are assembled in the two upper levels. Each of these levels renders a separate scheduling problem, and different levels are intertwined by finish-to-start precedences. For instance, the first assembly level receives components from the upper levels and it can be modeled as a *Flexible Flow Shop Problem*, where operations at each stage can be processed on one of several identical parallel machines. The third and uppermost level of the assembly is essentially a *Parallel Machine Scheduling Problem* with identical machines. In addition, to perform any assembly operation, both a machine and a worker are required, where the number of workers is typically smaller than the number of machines, making the former a *scarce resource*, similar to *renewable resources* in a Project Scheduling Problem.

As far as we know, no prior publications have addressed this problem. Therefore, we systematically tackle the problem by proposing a MILP model, lower bounds, and constructive as well as meta-heuristic approaches.

Finally, we extend our research to last-mile delivery problems, which are loosely related to scheduling problems but are becoming popular due to the increasing demand of e-commerce and the strategic integration of drones for sustainable deliveries. Specifically, we tackle a variant of a problem originally proposed in [21], named *Parallel Drone Scheduling Traveling Salesman Problem*. In this variant [22], a heterogeneous fleet composed of a truck and multiple drones collaborate to deliver parcels from a central depot. The truck behaves as a standard traveling salesman while drones have to perform back-and-forth trips from the depot and can be modeled as identical parallel machines. Similarly to scheduling problems, the objective is to minimize the makespan, i.e., minimize the longest route of vehicles. A unique aspect of this problem involves the *cooperative behavior of drones*, capable of connecting and working together

for specific deliveries. This collaborative approach enhances drones by boosting lifting capabilities and enabling faster delivery, thereby addressing limitations observed in scenarios where drones operate independently. However, this collaborative behavior makes the scheduling of drones a more complex task, requiring careful decisions on when the synchronization of multiple drones is advantageous, considering the cost of this operation and its subsequent benefits.

Due to the similarities of this mixed problem with scheduling ones, like the makespan objective and drones behaving as parallel machines with synchronizations, we investigate the effectiveness of state-of-the-art scheduling techniques for dealing with this problem. Specifically, we propose new Constraint Programming approaches, which are demonstrated to be among the best resolution methodologies for many shop scheduling variants, see, e.g., [23]. Furthermore, we extend the problem by considering multiple trucks, rather than a single one, thus introducing the *Parallel Drone Scheduling Vehicle Routing Problem* with collective (collaborative) drones.

The remainder of this work is organized as follows. In Section 2, we introduced preliminary concepts necessary to understand parts of this work, such as deep learning methodologies and the Job Shop. Section 3 outlines a novel supervised methodology for enhancing meta-heuristics and potentially other optimization methods for the JSP. Section 4 describes a new self-supervised methodology for training superior ML-based constructive heuristics for the JSP. Section 5 presents a new complex shop scheduling problem encompassing precedence and workforce constraints, as well as lower bounds and resolution approaches. Section 6 describes forms of a mixed problem, requiring the routing of vehicles and the scheduling of drones, along with constraint programming models to tackle two versions of this problem. Lastly, Section 7 closes with final considerations.

2 Preliminaries

This section is meant to provide a concise overview of key concepts employed in this work and elucidate the rationale behind chosen research directions. Notably, we introduce relevant Machine Learning paradigms, types of neural networks, and formally describe the Job Shop Scheduling Problem. Additionally, we review methodologies used to address this scheduling problem.

2.1 Machine & Deep Learning

Machine learning, and particularly deep learning, has become ubiquitous in many fields. In the last years, there has been a growing interest in applying learning techniques to Combinatorial Optimization (CO) problems [24]. The rationale is that CO problems are solved daily, and one can use ML to learn better algorithms from the thousands of instance-solution pairs produced. Theoretically, once a ML model has learned valuable pieces of information about a problem, the model can be used for either solving or aiding the resolution of large and challenging instances.

However, applying ML to CO problems is far from being trivial as the construction of solutions often involves long sequences of decisions subject to multiple constraints. Designing the right model and learning from these long constrained sequences are difficult tasks with traditional ML methodologies. For instance, a natural idea for creating neural solvers is to learn how to make decisions from (near-)optimal solutions of past or small synthetic instances. However, having access to tens of thousands of optimal solutions is unusual, and generating them is extremely costly. Moreover, some CO problems like the JSP are multimodal, meaning there might be multiple and different optimal solutions for an instance, and learning a model that approximates a one-to-one instance-solution mapping may pose additional obstacles.

Therefore, in the remainder, we describe learning paradigms that may answer some of the challenges posed by CO problems or have proved effective on various kinds of packing, routing, and scheduling problems. Specifically, we present common Reinforcement Learning approaches to deal with CO problems and describe other two ML paradigms for reducing labeling costs which are much

less investigated, if not at all. Lastly, we outline a well-known architecture for dealing with CO problems, namely the Pointer Network.

2.1.1 Reinforcement Learning

Reinforcement Learning (RL) is one of three basic ML paradigms, alongside supervised and unsupervised learning. Reinforcement Learning is concerned with how an agent ought to take actions in an environment in order to maximize a cumulative reward [15]. Different from supervised learning, RL agents learn from experience (the cumulative reward) derived by directly acting within the environment. Therefore, no labeled input-output pairs are required nor sub-optimal actions have to be explicitly corrected. *This unique characteristic makes RL a very appealing paradigm wherever generating labeled pairs is challenging and expensive, like in combinatorial optimization problems.*

In RL, an agent takes actions within an environment and how this environment reacts is defined by a model. Although the model of the environment is sometimes available, in which case a Dynamic Programming approach is often the way to go, RL agents commonly learn without this model (*model-free RL*) or try to explicitly learn it as part of the algorithm. Within the environment, the agent can stay in one of many states and choose actions from a defined pool to switch to other states. Once an action is taken, it is the environment that returns the new state and a reward as feedback.

The agent follows a behavioral *policy*, either a deterministic or stochastic function, that tells which action to take in any state of the environment. During the learning process, the agent accumulates knowledge about the environment, adjusts its policy, and makes new decisions on which action to take next so as to efficiently learn the best policy. Associated with a policy, there is always a *state-value function* that measures how rewarding a state is in terms of future rewards, a.k.a. *return*. In other words, the state-value function gives the expected return the agent receives from a state by acting according to the current policy. Similarly, the *action-value function* is used to define the goodness (still in terms of expected future rewards) of being in a certain state and taking a specific action. The concepts of policy, state-value, and action-value functions are used to learn the best (sometimes optimal) behavior for a given environment. We refer the reader to [15] for an in-depth treatment of these concepts.

As reviewed in [25], the two broad families of RL algorithms mainly adopted for CO problems are: *value-based* and *policy-based* methods.

Value-based methods focus on finding a policy through the approximation of the state-value and action-value functions. Once the value function is fully known, meaning that the optimal function has been extracted for an environment, an optimal policy can be found by acting greedily with respect to such a function. For instance, when the state-value function is known, one can find optimal actions with a greedy one-step search: picking the actions that result in the next state with the maximum value. Therefore, approximating the best, and possibly the optimal, value function is the key of value-based methods.

The simplest value-based methods are Monte Carlo ones. The idea of these methods is to learn from episodes of raw experience. Once many episodes have been completed, i.e., many solutions of a CO problem are constructed, the return of a state can be approximated as the discounted mean of future rewards received in episodes visiting the state. Monte Carlo methods learn only on the completion of episodes, therefore, other methods focus on learning while playing episodes. One of the most popular methods of this type is Q-learning [15], and its deep variant Deep Q-learning [26]. In Q-learning, the action-value function is iteratively updated after visiting a state by learning from the received reward.

With the rise of deep learning, researchers have started to explore the potential of using neural networks to approximate action-value functions. However, Q-learning is demonstrated to suffer from instability and divergence when combined with a nonlinear function approximation [25, 15, 27]. Therefore, various fixes have been introduced to reduce training instability, including experience reply and its prioritize version [27]; periodic updates; and double dueling networks [28]. Although the effectiveness of these fixes is largely demonstrated, they introduce further complexity and hyper-parameters to be tuned.

In contrast to value-based, **policy-based methods** attempt to directly find the (optimal) policy, often parametrized by neural networks, without the need to approximate value functions explicitly. These methods, rooted in the policy gradient theorem [29, 15], collect experience (rewards) in the environment using the current policy and then optimize it based on this experience. A vanilla policy gradient method is REINFORCE [15] (a.k.a. Monte-Carlo policy gradient) which estimates returns through Monte-Carlo methods, updating the policy with the gradient descent algorithm. A widely used variation of REINFORCE is to subtract a baseline value, such as the average reward over the sampled episodes, from the return to reduce the variance of gradient estimations.

Instead of using simple statistics over the sampled rewards, one can use neural networks to approximate value functions, effectively going in the direction of hybrid policy- and value-based methods. Knowing the value function can assist the policy updates, like reducing gradient variance in vanilla policy methods, and that is exactly what the Actor-Critic method does. Actor-Critic methods [30] consist of two models: a Critic approximating the value function and an Actor shaping the policy. By using the estimates of the expected return given by the critic, the updates of the actor are less noisy. However, the critic has to be updated as well like in value-based methods.

Policy-based methods typically suffer from high variance in gradient updates, sample complexity, and sensitivity to hyperparameters [25, 31, 32]. Proximal Policy optimization algorithms [32] aim to address these issues by improving robustness to hyper-parameters and constraining policy updates to remain near (within a trusted region) the old policy. Whereas, [31] focuses on asynchronously executing multiple policies (actors) in parallel to sample more uncorrelated rewards and improve the training, similarly to mini-batch stochastic gradient descent [33]. As in value-based methods, these effective fixes add complexity to both understanding and implementing these methods.

While Reinforcement Learning has proven to be a valuable paradigm for training neural solvers, its widespread understanding is still limited, and its complexity cannot be overlooked. Given these factors, and considering that supervised approaches are relatively less explored for combinatorial problems, our primary focus in this work is on supervised learning methodologies.

2.1.2 Semi- and Self-Supervised Learning

In the previous Section 2.1.1, we briefly introduced RL approaches for tackling CO problems and motivated them by stating that they remove the need for expensive labels. However, RL is not the only paradigm that removes or reduces the need for labeled pairs while training ML models.

Semi-Supervised learning leverages both labeled and usually a large amount of unlabeled data to train neural networks with fewer annotations [19]. Examples of these approaches are: Feature Extraction methods [34] that pre-train models in an unsupervised manner to extract representative features before fine-tuning on labeled data; Clustering approaches [35] that group labeled and unlabeled data into clusters and propagate labels within clusters to aid supervised signals; and Regularization approaches [36, 37] that add a term to a supervised loss for improving generalization. Despite these approaches, the most common Semi-Supervised approach involves iteratively training models using labeled data alongside previously unlabeled data, the latter being annotated with the most confident predictions resulting from previous iterations [19]. Notably, this process progressively enhances models until everything is labeled, where newly annotated data is termed *pseudo-labeled* data [38]. Many variants of this pseudo-labeling process exist, mainly differing in the way models are retrained with the new pseudo-labels, e.g., Self- and Co-Training [39].

Although some of these approaches are extensively investigated in fields like computer vision and natural language processing, we could not find applications to combinatorial problems. As labeling is very costly in general, and even more in combinatorial problems, we believe that applying semi-supervised approaches to these problems might be an interesting and promising research direction.

Another popular ML paradigm that may help either in reducing or removing labeling costs is **Self-Supervised** learning. The goal of Self-Supervised learning is to reduce the need for annotations by learning meaningful and generic patterns in unlabeled data with *pretext* tasks [18], i.e., artificial and unsupervised tasks nevertheless related to downstream applications. The advantage of pretext tasks is to simplify subsequent supervised tasks by mainly requiring models to associate patterns with targets. Self-Supervised approaches are commonly divided into *contrastive* and *predictive* [40].

Contrastive approaches encode inputs into latent vectors by reducing the similarity from positives (augmented copies of the input) and increasing that from negatives (different inputs). Two examples are MoCo [41] and SimCLR [42], which showed the importance of creating hard positives and negatives. Another (contrastive) example is DeepCluster [43] which uses K-Means to group simi-

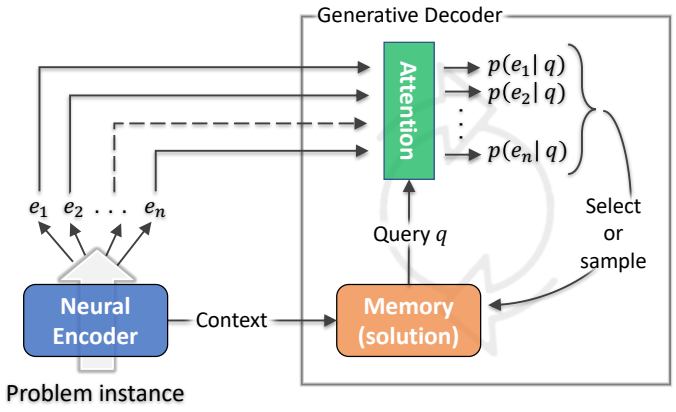

Figure 1: The operations of the Pointer Network. Initially, the neural encoder takes in the problem instance and generates an embedding e_i for each item i. Then, the generative decoder uses these embeddings, contextual information about the instance, and the already selected items to iteratively generate solutions in the form of permutations. The probability $p(e_i \,|\, q)$ of selecting an item i is generated at every iteration with an attention module that combines the embeddings and the query q produced by the memory module.

lar latent vectors and train a model in predicting their *pseudo-labels*, i.e., the indices of the clustering stage.

Predictive approaches focus on (re)constructing a desired output from a potentially altered input without requiring similarity metrics. Examples are Denoising and Variational Auto-Encoders [33], which use various strategies to encode inputs into latent vectors and decode them back into the original inputs, or Masked Auto-Encoders [44, 45], where models are trained to reconstruct masked patches of the input from unmasked ones.

Despite the increasing adoption and the potential of these approaches [18, 40], they have little to no applications for combinatorial problems. One obstacle limiting the application of contrastive approaches is the lack of augmentations that preserve the solution space of instances [46], without them is hard to apply constructive strategies. Moreover, applying predictive approaches requires identifying meaningful patterns and designing effective pretext tasks, both of which are non-trivial operations in combinatorial problems.

2.1.3 The Pointer Neural Network

The *Pointer Network* [47] (PN) is an encoder-decoder architecture inspired by natural language processing models and commonly applied to solve different CO problems [47, 48, 49, 50]. The main reason for employing the PN in problems such as the Traveling Salesman Problem and the Job Shop Problem is that they can be solved by defining one or multiple permutations of their items. Standard

convolutional or feedforward neural networks cannot approximate sorting functions, as sorting is not a trivially derivable operation. To fix this issue, the PN was proposed to learn functions for sorting a variable number of input items.

More in detail, the PN constructs a permutation of the input items in an autoregressive (iterative) manner, where at each construction step it first generates a score for all the items not yet inserted and selects one based on these scores. After each step, the scores of remaining items – not yet in the permutation – are regenerated and the process is repeated until the permutation is complete. As shown in Figure 1, this iterative process is realized with two separate neural networks: a *neural encoder* and a *generative decoder*.

The role of the encoder is to embed a problem instance into the multidimensional space where the decoder works. The encoder considers patterns or characteristics of the items within an instance and produces meaningful embedding representations e_i for all its items. Although this is not always the case, the embedding generation step is normally executed once, before starting the construction of permutations, and prepares the ground for the decoder. Other than producing embeddings, the encoder often produces the context necessary to initialize the decoder, like a graph embedding or a state for recurrent networks.

The decoder uses the embeddings e_i and internal information derived from past selections to generate scores and select items. Logically, the decoder can be divided into two parts: (i) a memory module that uses the instance context (if provided or necessary) and the partially constructed permutations to generate a query q; (ii) an attention module that receives in input the variable number of items and the query q to produce a probability $p(e_i \mid q)$ of selecting each item i. Once the probabilities have been generated, many different strategies can be used to select the next item [51, 52, 53, 14, 54]. Some common strategies are selecting the item with the highest probability (*greedy* decoding) or sampling one with a probability of $p(e_i \mid q)$ (*sampling* decoding). Note that the decoder is autoregressive as it updates the information contained in the memory after each selection and generates from it the queries for future iterations. In this update step, also the embeddings e_i are sometimes regenerated [13].

Overall, the PN is a flexible architecture that can comprise different types of neural networks. For instance, [47] used a Recurrent Neural Network (RNN) for both the encoder and decoder; [48] replaced the RNN encoder with element-wise projections; [49] introduced a Transformer inspired encoder; and [50] augmented the RNN encoder with a Graph Neural Network [55]. Note that most of these variations focus on the encoder part, while the decoder is typically based on the alignment score produced with either additive [56] or dot-product [57] attention. In addition, this architecture is robust to different training regimes, e.g., [47] trained the PN in a purely supervised framework while [51, 50, 49, 58] used various RL algorithms.

2.2 Job Shop Scheduling Problem

In the Job Shop, we are given a set of jobs $J = \{J_1, \ldots, J_n\}$, a set of machines $M = \{M_1, \ldots, M_m\}$, and a set of operations $O = \{1, \ldots, o\}$. Each job $j \in J$ comprises a sequence of $m_j \in \mathbb{N}$ consecutive operations $O_j = (l_j, \ldots, l_j + m_j) \subset O$ indicating the order in which the job must be performed on machines, where l_j is the index of the first operation of the job ($l_j = 1 + \sum_{i=1}^{j-1} m_i$). An operation $i \in O$ has to be performed on machine $\mu_i \in M$ for an uninterrupted amount of time $\tau_i \in \mathbb{R}_{\geq 0}$. Additionally, machines can handle one operation at a time. The objective of the JSP is to provide an order to operations on each machine, such that the precedences within jobs are respected, the operations do not overlap on each machine, and the time required to complete all jobs (*makespan*) is minimized. In scheduling theory [3], this problem is identified as $Jm \,||\, C_{max}$.

Let $\pi = \{\eta_1, \ldots, \eta_m\}$ be a solution of an instance, and $\eta_k = (s_1^k, s_2^k, \ldots, s_{n_k}^k)$ the permutation or sequence of $n_k \in \mathbb{N}$ operations on machine $k \in M$. The permutation η_k fixes the order of operations on machine k, and s_t^k, with $t \in \{1, \ldots n_k\}$, gives the operation of some job j that is processed in the t^{th} position. Lastly, we use $C_i(\pi)$ to identify the completion time of operation $i \in O$ in π.

2.2.1 Disjunctive Graph

It is common to represent a JSP instance as a **disjunctive graph** [59] $G = (V, A, E)$ (see left of Figure 2), where:

- V contains one vertex for each operation $i \in O$ with weight τ_i;
- A is the set of *conjunctive* (directed) arcs connecting consecutive operations of jobs, these arcs reflect the order in which operations must be performed to complete jobs;
- E is the set of *disjunctive* (undirected) edges connecting those operations to perform on the same machines.

When the JSP is represented as a disjunctive graph G, finding a solution means providing a direction to all the edges in E, such that the resulting graph is directed and acyclic (all precedences are respected), and its weighted longest path (the *critical path*) is minimized. We will refer to the set of oriented edges with $\hat{E}$ and the corresponding digraph with $\hat{\mathcal{G}} = (V, A, \hat{E})$. We remark that there is a unique one-to-one correspondence between a generic solution π_g and a digraph $\hat{\mathcal{G}}_g$, as orienting the edges of E is equivalent to creating permutations $\{\eta_k\}_{k=1}^m$ and vice versa. In Figure 2, we better highlight the unique correspondence between a solution and its digraph. Lastly, we remark that any feasible solution of a JSP instance results in an acyclic digraph. Therefore, proving the feasibility of a solution is equivalent to checking whether its corresponding digraph is acyclic [3].

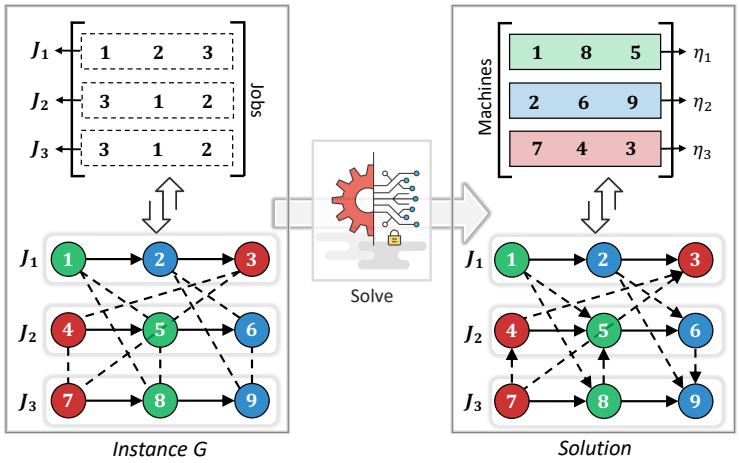

Figure 2: On the left, an example of a disjunctive graph that represents a JSP instance with 3 jobs (J_1, J_2, J_3) and 3 machines (dashed lines). On the right, a feasible solution that gives the sequence of operations η_k for each machine k.

2.2.2 Constructing Solutions

JSP solutions can be constructed step-by-step as a *sequence of decisions*, an approach adopted in constructive heuristics such as Priority Dispatching Rules [11, 3]. This construction process can be visualized using a branch-decision tree, shown at the bottom of Figure 3. Each path from the root node (R) to a leaf node presents a particular sequence of $o = |O|$ decisions and leads to a feasible solution, such as the one highlighted in gray.

At every node in the tree, one uncompleted job needs to be selected, and its ready operation is scheduled in the partial solution being constructed. Due to the constraints imposed by jobs (i.e., the conjunctive arcs) and the partial solution π_t constructed up to decision t, there is at most one operation $o(t, j) \in O_j$ that can be scheduled for any job $j \in J$. Thus, by selecting a job j, we uniquely identify an operation $o(t, j)$ that will be scheduled by appending it to the partial permutation of its machine. Once a job is completed, it cannot be selected again, and such a situation is identified with a cross in Figure 3.

We also highlight there exist multiple paths in the branch-decision tree that lead to the same solution. This indicates that certain JSP solutions are more probable than others. In some instances, due to the distribution of processing times τ_i, optimal or near-optimal solutions may be found among the less likely paths, making the construction of quality solutions a more complex task.

This observation holds significant implications. For instance, it might be more difficult for constructive heuristics to create (near-)optimal solutions in instances where these solutions result from very unlikely paths. Additionally, some ML approaches (especially RL ones) use the concept of instance complexity to learn first from easy instances and move to more complex ones later on in the

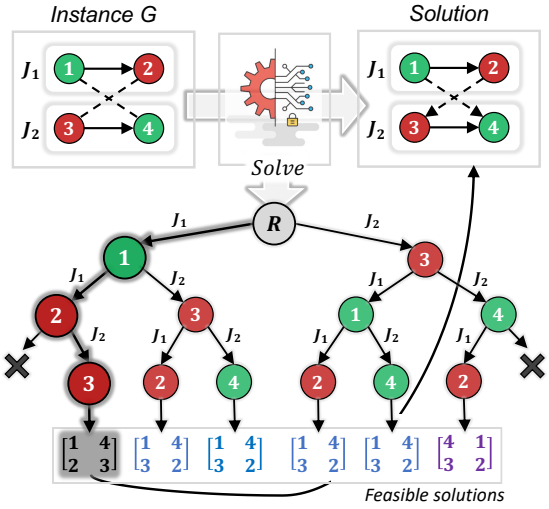

Figure 3: The sequences of decisions that allow constructing solutions for a JSP instance with two jobs and two machines (identified with different colors).

training, something known as curriculum learning [60, 61]. In these curriculum learning strategies, the complexity is often associated with the instance size only, see e.g., [14]. Our observation can be used to further improve these approaches.

One of the most common constructive heuristics in scheduling is the Priority Dispatching Rule (PDR). PDRs [11] are simple heuristics that assign operations to machines based on job priorities, the higher the job priority more likely is to schedule its ready operation. In general, priorities are computed with hard-coded rules that consider the status of the schedule and characteristics of jobs, machines, and operations. Designing an effective PDR is difficult and requires substantial domain knowledge, especially on complex problems like the JSP. Moreover, as partially explained in the above complexity observation, the performance of a PDR may drastically vary on different instances. Therefore, in the last years, researchers tried to enhance and automate the design of superior PDRs with the help of Machine Learning [13, 62]. In Section 4, we contribute in this direction.

2.2.3 Improving Solutions: Meta-heuristics

As constructive heuristics are in general unable to deliver high-quality solutions, iterative methods have been proposed to improve the quality of JSP solutions. Among others, meta-heuristics are the most common and effective ones.

At a high level, meta-heuristics are strategies that guide an underlying, more problem-specific heuristic, to provide a sufficiently good solution to an optimization problem. As meta-heuristics make relatively few assumptions about the problem being solved, they can be used for a variety of different problems [63].

The goal of a meta-heuristic is to find (near-)optimal solutions by efficiently exploring a subset of the solution space that is otherwise too large to be completely enumerated. Therefore, compared to certain algorithms, meta-heuristics do not generally guarantee that a globally optimal solution can be found or can be proved as such.

The literature on meta-heuristics is rich and several books and surveys have been published on the subject [63, 64, 65]. While the field features high-quality research, e.g., various works provide formal theoretical results on the possibility of finding global optima [63, 6, 7], many publications have been of poor quality. Common issues include implementation flaws, presentation vagueness, poor experiments, and ignorance of previous literature [66]. Therefore, selecting an effective meta-heuristic and correctly implementing it for efficiently solving real-life problems requires a substantial knowledge of the field.

In general, a meta-heuristic will be successful on a given problem if it can provide the right balance between diversification (exploration) and intensification (exploitation). This balance is important, on one side to quickly identify regions of the space with high-quality solutions, and on the other side to avoid wasting a lot of time in already explored or poor quality regions. The main differences between the most popular meta-heuristic frameworks concern the particular way in which this balance is achieved.

Some meta-heuristics can be seen as "intelligent" extensions of local search algorithms. The goal of this kind of meta-heuristic is to escape from local minima to proceed the exploration of the solution space for hopefully finding better minima. This is for example the case of Tabu Search [67], Iterated Local Search, Variable Neighborhood Search, and Simulated Annealing [5, 6]. These meta-heuristics, also called trajectory-based algorithms, work on one or several neighborhood structures. In a nutshell, they describe trajectories in the solution space of an instance, starting from initial solutions and visiting neighbor solutions according to some criteria [68, 69]. Each trajectory generally stops either when no improving solution exists in the neighborhood, i.e., the current solution is a local optimum, or when a predefined criterion is met.

A different philosophy can be found in Ant Colony Optimization [9] and Evolutionary algorithms [70, 71]. They incorporate a learning component in the sense that they implicitly or explicitly learn correlations between decision variables to identify high-quality regions of the solution space. This kind of meta-heuristic performs a sort of biased sampling of solutions. For instance, in Evolutionary algorithms this is achieved by recombining certain parts of selected solutions, and in Ant Colony Optimization by sampling the solution space in every iteration according to a probability distribution.

Despite the kind of meta-heuristics, just picking one or another meta-heuristic does not guarantee to successfully tackle a problem. The effectiveness of every meta-heuristic depends on a brittle and complex balance of its elements, like the neighborhood structure, the searching procedure, and other mechanisms. This balance is achieved by specifically designing elements on the faced problem, such that the algorithm can intensify promising regions while escaping from local optima. Therefore, selecting, designing, and assembling the right elements

is extremely important and requires domain and algorithm-design expertise.

In the context of the JSP, the most successful type of meta-heuristics are trajectory-based ones, like Tabu Search and Simulated Annealing [6, 72]. The breakthrough work in the field of meta-heuristics is [6]. This work adapted the Simulated Annealing (SA) [5] and proposed one of the most studied and effective neighborhood structures for the JSP, called N1. N1 was the first to show how it is possible to construct the neighborhood of a solution without incurring in unfeasible solutions. In addition, it guarantees the existence of a trajectory that leads to global minima, the so-called *convergence property*.

After this work, many variations and extensions of N1 were proposed in [72, 7, 73], mostly in the context of a Tabu Search [67]. The most successful application of the TS to the JSP is [7], where the authors proposed a reduced variation of N1 in which some of the neighbor solutions were removed since they cannot immediately improve the current solution. In [7] is also proposed the best implementation of the TS for the JSP, successively refined in [74] by incorporating elements of path relinking in the generator of initial solutions.

Besides the TS and SA, there are other meta-heuristics proposed to tackle the JSP [9, 75, 8]. In these regards, we just want to stress that regardless of the meta-heuristics, e.g., single-source or population-based [68], an ad-hoc searching procedure or a local search is often required to enhance performance [9, 75, 8].

As creating new effective meta-heuristics for the JSP, its variations, and similar scheduling problems is a challenging and time-consuming process, in Section 3, we propose a novel methodology that may benefit in these regards.

2.2.4 Machine Learning Applications

Supervised.

The Job Shop Scheduling has been tackled in a *fully supervised* manner by mostly relying on exact methods such as MILPs. The idea of these methods is to create small synthetic instances, compute their (near-)optimal solutions, and learn from them scheduling patterns that generalize to larger instances.

One of the first applications of ML is presented in [62], where a neural network selects the most suited PDR from a pool to schedule the next operation. The decisions of the neural network are based on the current system state and the training is done through simulations. Another approach for enhancing PDRs is proposed in [76], where imitation learning [24] is leveraged to learn superior dispatching rules from optimal solutions. This work demonstrated how learning from optimal solutions is not enough to produce robust PDRs. In [77], the authors proposed to jointly use optimal solutions and Lagrangian terms while training to better capture the JSP constraints. A different supervised proposal is in [12]; herein, the MILP generates the likelihood of machine permutations being in an optimal solution, and a Recurrent Neural Network [33] learns to predict the likelihood of permutations for guiding a meta-heuristic.

Despite the potential demonstrated by these supervised approaches, they rely on optimality information which is expensive to generate and limits their adoption and widespread.

Reinforcement.

Therefore, several recent works focus on adapting (Deep) Reinforcement Learning approaches. The advantage of RL is that it does not need costly optimal or near-optimal solutions, but only requires a correct formulation of the Markov Decision Process [15], as explained in Section 2.1.1. After formulating the Markov Decision Process, a policy to either schedule operations or make decisions is learned from the experience derived by resolving the same instances many times. RL has been mainly applied to create superior PDRs while only a few works used it inside a meta-heuristic.

Regarding research efforts focusing on PDRs, [78] proposed an actor-critic architecture [30], where the critic evaluates the value of decisions in the partial schedule, and the actor learns to make decisions based on the schedule and the critic estimations. In [13], an actor-critic is also proposed, but with a Graph Neural Network (GNN) [55] to construct an adaptive representation of the partial schedule. One interesting aspect of this work is that the authors underline how GNNs seem to have poor performance when applied to disjunctive graphs. Another example of an actor-critic architecture coupled with a GNN is in [79]. This work specifically designed a GNN architecture for encoding the disjunctive graph representing JSP instances by using a rich set of features to describe operations. Differently from these proposals, [58] did not use any GNN but a rather complex Transformer-based [57] model, without substantially improving performance. We underline that all these works adopted Proximal Policy Optimization (PPO) algorithms [32] to train their models, one may therefore conclude that such algorithms are necessary when dealing with the JSP. However, a remarkable non-PPO proposal is in [80], where a Double Deep Q-Network is equipped with prioritized Experience Replay [27] to schedule job operations, proving that also value-based approaches are indeed effective (see Section 2.1.1 for more information on RL algorithms). More recently, [14] proposed a curriculum learning strategy that gradually increases the size of JSP instances to increase complexity, still in the context of an actor-critic algorithm.

Note that all these works construct JSP solutions step-by-step as explained in Section 2.2.2, using reward signals that are only indirectly related to the makespan, such as greedily minimizing the makespan increment of partial solutions [13] or minimizing the number of waiting jobs [79]. We argue that such local rewards may sometimes result in sub-optimal performance for learning to construct globally optimal solutions. Thus, in Section 4, we propose a new training strategy that focuses on the objective of complete solutions.

As reviewed in [81], ML can be fruitfully integrated in the most common meta-heuristics and constitutes an opportunity to enhance, simplify, and automate the creation of effective algorithms. Some examples of how ML techniques can be integrated into meta-heuristics for scheduling problems are [82, 83, 84].

In [82], it is proposed a DRL-based rewriting method in which a region-picking policy selects regions of solutions that are rewritten with rules selected by a rule-picking policy. Picking the right regions and selecting the best rewriting rule are non-trivial operations, and learning to perform them from experi-

ence outperformed heuristic rules. The authors of [83] identified three points of interventions within meta-heuristics where better strategies can be learned. Thus, they used a Double Deep Q-learning approach to learn when to accept new candidate solutions, which among different neighborhoods has to be used, and when is necessary to apply a perturbation before continuing the exploration. Meta-heuristics enhanced in these three points achieved slightly better performance than non-enhanced ones on benchmark instances. Lastly, in [84], a Variable Neighborhood Search is enhanced with a mechanism that favors the creation of solutions having promising attributes during the shaking step. Although this work does not use any ML techniques, learning to construct these solutions might be a viable and better approach. For other examples of how to combine ML with meta-heuristics, we refer the reader to [81].

Recently, [85] presented a hybrid imitation learning and policy gradient approach coupled with Constraint Programming (CP) for outperforming PDRs and a CP solver. This last work unveils another way to combine RL for enhancing JSP resolution methods.

Despite these premises, meta-heuristics did not receive the same attention as PDRs in hybridization with ML for the JSP. Therefore, in Section 3, we contribute in this direction.

3 Enhancing Tabu Search for the Job Shop Problem

The *Job Shop Scheduling Problem* [3] is a notorious NP-hard combinatorial problem with many practical applications in industry [1, 2]. Mixed Integer Linear Programming and Constraint Programming are two optimization methods that can solve the JSP [4] and prove the optimality of solutions, valuable information for both academia and industries. Although these methods are becoming every day faster, they do not scale well on medium and large instances [4], and they become rapidly useless even in small but complex industrial environments [1]. For these reasons, approximation methods are still largely employed and constitute an active area of research.

Meta-heuristic algorithms are the state-of-the-art approximation method when comes to solving the JSP. Once implemented, a meta-heuristic can be tuned to rapidly improve solutions (in a matter of minutes) or can be left running for a long time to find the best solution possible. However, designing and implementing effective meta-heuristics is a long and challenging process, as largely reviewed in Section 2.2.3. Sometimes, even when effective meta-heuristics have already been designed, it might be difficult to reproduce results or adapt algorithms to variations of the original problem.

As reviewed in [81], Machine Learning can be fruitfully integrated into common meta-heuristics and constitutes an opportunity to enhance, simplify, and automate the creation of effective (meta-)heuristics. For instance, ML can learn how to make effective decisions, how to recreate partially deconstructed solutions, and how to avoid exploring unpromising regions of the solution space. Despite these premises, meta-heuristics did not receive the same attention as PDRs in hybridization with ML, and we believe there is much to gain from such a combination. Therefore, we try to fill this gap by enhancing the effectiveness and automating the creation of meta-heuristics thanks to ML approaches.

To this end, we focus on supervised learning as adopting RL makes the training a more challenging task, it is not easy to reproduce [16], and takes a lot of time [13], especially for Monte Carlo-based methods [15]. However, finding which type of information might help enhance meta-heuristics is not trivial. Moreover, designing a supervised methodology requires special care and

a high-quality dataset to effectively learn valuable information once identified.

This is particularly true in the JSP, as not all the solutions to an instance are feasible, i.e., respect the problem constraints, and for those feasible, the objective value (e.g., the makespan or the total tardiness) is not trivially derivable. For these reasons, the application of supervised learning to the JSP requires a learning task tightly related to the objective function, and that may fit in the back-propagation algorithm. We thus propose as a novel supervised learning task to *learn the quality of a machine permutation, i.e., how good is the sequence of operations on a machine*.

Understanding whether a sequence of operations on a machine is of high quality is a valuable but difficult task in the JSP [3]. Since having a method to judge machines is important, either for speeding up existing algorithms or even in machine-based decompositions, we present an original methodology to learn the quality of machines by means of *sequential deep learning* and a *MILP solver*. There already exists in literature approaches to evaluate a machine, most notably [17], but they frequently estimate the criticality of machines, a related but different concept. Contrary, we define the quality of a machine permutation as the *likelihood of finding this permutation in an optimal or near-optimal solution*.

We evaluate the impact of our proposal by comparing the results obtained with one of the best meta-heuristics for the JSP, namely the Tabu Search (TS), with and without these quality estimations. In addition, we compare the results of the TS with some of the DRL approaches to justify our proposal for enhancing existing approximation algorithms.

Note:

An extracted version of the work published at:
Andrea Corsini, Simone Calderara, and Mauro Dell'Amico, "*Learning the Quality of Machine Permutations in Job Shop Scheduling*", in IEEE Access, Volume 10, 2022. DOI: 10.1109/ACCESS.2022.3207559.

3.1 Learning the Quality of Machine Permutations

Our novel supervised learning task about the JSP is to *predict the quality of machine permutations, where the quality is the likelihood of finding this permutation in an optimal solution*. To justify why our learning task should help in solving the JSP, we briefly report the intuition behind the proof of the convergence property of the N1 neighborhood (see [6] for the complete proof).

Let π_1 and π_o be respectively a feasible and an optimal solution of an instance. The converge property implies that from any π_1, it is possible to construct a trajectory of solutions through N1 that allows moving from π_1 to an optimal solution π_o. The proof starts from the definition of a special set of

critical arcs (remember that critical arcs are those in the longest path of $\hat{\mathcal{G}}$):

$$K_1(\pi_o) = \{(v,w) \in \hat{E}_1 \,|\, (v,w) \text{ is critical} \wedge (w,v) \in \hat{E}_o\} \tag{1}$$

that is the set of critical arcs in $\hat{\mathcal{G}}_1$ that do not belong to the optimal solution $\hat{\mathcal{G}}_o$, where $\hat{\mathcal{G}}_1$ and $\hat{\mathcal{G}}_o$ are the acyclic digraphs associated to π_1 and π_o respectively. When $\pi_1 \neq \pi_o$, this set is always non-empty, and it is possible to create a finite trajectory $(\pi_1, \pi_2, \ldots, \pi_o)$ that guarantees to reach an optimal solution, where π_2 is obtained from π_1 by reversing an arc in K_1. Clearly, the convergence is a desirable property for a neighborhood structure, but in practice, it is of no help as it requires knowing the set K of critical arcs to reverse.

Nevertheless, this proof leads us to what might be beneficial for solving the JSP: *an information about which critical arcs are unlikely to be in an optimal solution*. At least in the context of N1, knowing this information allows excluding those solutions that introduce arcs unlikely to be in $\hat{E}_o$, resulting in better exploration and a faster convergence towards optima. However, there is a problem in learning a function that gives the likelihood of finding an arc in an optimal solution: the representation of the arc must encode enough information about the entire solution.

Instead of learning this function, we propose to learn a function that receives in input the machine permutation associated with an arc and outputs the likelihood of finding this permutation in an optimal solution. If the permutation resulting from the inversion of a critical arc is of higher quality than the original permutation, the reversed arc has a higher chance of being in $\hat{E}_o$. Therefore, learning such a function still allows discriminating which critical arcs should be reversed. In addition, it simplifies the learning task since permutations intrinsically encode more information about solutions than single arcs.

Based on this theoretical intuition, our learning task should help solve the JSP in at least those approximation algorithms based on N1. Note that the proposed learning task might also benefit other approximation methods, e.g., machine-based decomposition and ruin-and-recreate algorithms [86], but we leave this analysis to future works.

What remains uncovered is how the quality y_k of a machine permutation η_k can be quantified. To define the quality y_k, we rely on the concept of makespan, and we compute:

$$y_k = 1 - \tanh\left(\frac{C_{max}(\eta_k)}{C^{opt}_{max}} - 1\right) \tag{2}$$

where $C_{max}(\eta_k)$ is the best makespan found by imposing η_k as part of the solution, C^{opt}_{max} is the optimal makespan of the instance, and tanh is the hyperbolic tangent function.

Note that Eq. 2, beyond giving the mathematical definition of the quality of a machine permutation, also points out the methodology needed to estimate this quality. This methodology includes a method to optimally solve the JSP and a method to find the best solution with an imposed sequence η_k. With these methods, Eq. 2 estimates y_k by comparing the best makespan found with the sequence η_k against the optimal makespan, and it scales this comparison with

the *tanh* function. When η_k is near-optimal, meaning that $C_{max}(\eta_k)$ is close to the optimal makespan, y_k takes a value close to 1. Contrary, when $C_{max}(\eta_k)$ is far from the optimal value, y_k takes a value close to 0. Due to its definition, the quality of a permutation is always a value in the interval $[0, 1] \subset \mathbb{R}$, thus, it can be interpreted as a kind of probability (or a likelihood parameterized by some parameters) of finding the permutation in an optimal solution.

As the method to optimally solve the JSP, we propose to use a MILP solver by formulating the problem as a disjunctive model [4]. As pointed out in [4], today solvers can solve instances with 10 jobs and 10 machines in a few seconds.

Instead, as the method to find the best makespan $C_{max}(\eta_k)$ by imposing a sequence η_k, we propose to use a modified version of the standard disjunctive model, again in a MILP solver. In this modified version, we introduce a set of constraints to prevent the solver from changing the order of the sequence η_k. Note that this modification effectively reduces the solution space and speeds up the solver. The modified disjunctive model is then:

$$
\begin{aligned}
\min \quad & C_{max}(\eta_k) && && (3)\\
\text{s.t.} \quad & x_i \geq x_{i-1} + \tau_{i-1} && \forall j \in J;\ \forall i \in O_j \setminus \{l_j\} && (4)\\
& x_i \geq x_t + \tau_t - Q\, z_{it} && \forall h \in M;\ \forall i, t \in \eta_h, i < t, && (5)\\
& x_t \geq x_i + \tau_i - Q\,(1 - z_{it}) && \forall h \in M;\ \forall i, t \in \eta_h, i < t && (6)\\
& x_{s_t^h} \geq x_{s_{t-1}^h} && \forall h \in M, t = 2, \ldots, n_h && (7)\\
& C_{max}(\eta_k) \geq x_{l_j+m_j} + \tau_{l_j+m_j} && \forall j \in J && (8)\\
& z_{it} \in \{0, 1\} && \forall h \in M;\ \forall i, t \in \eta_h && (9)\\
& x_i \geq 0 && \forall i \in O && (10)
\end{aligned}
$$

The model has two decision variables: x_i gives the starting time of operation $i \in O$, and, z_{it} takes value 1 if operation i precedes operation t on their machine. The set of constraints (4) guarantees that for each job, the start time of every operation must be equal to or higher than the completion time of the previous operation. The disjunctive constraints in sets (5) and (6) guarantee that the start time of an operation i must be higher than the completion time of another operation t when i is scheduled before t and vice versa. Finally, the set of constraints (7) fixes the order of operations on machine h to be equal to $\eta_h = (s_1^h, s_2^h, \ldots, s_{n_h}^h)$, and the set (8) computes the makespan. The value of Q is set to $\sum_{i \in O} \tau_i$ to ensure the correctness of the disjunctive constraints.

Summarizing, the methodology to obtain the quality of a machine permutation η_k starts by optimally solving the JSP instance, then the best makespan $C_{max}(\eta_k)$ is found with the presented modified disjunctive model, and finally, the quality is computed with Equation 2.

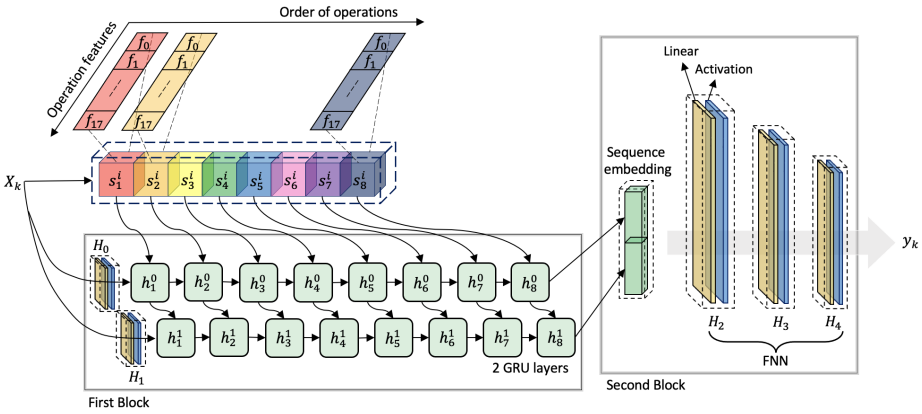

Figure 4: The architecture of the oracle. On the left, the 2-dimensional representation X_k of a sequence is transformed into a sequence embedding through the first block. In the right, the sequence embedding is fed into the second block to compute the quality y_k.

3.2 The Neural Network Oracle

To predict the quality y_k of a sequence η_k, we designed a sequential deep learning model that is sensitive to the order of the input. We will refer to such a model as the *oracle*.

As a standard in sequential deep learning, each operation of a sequence $\eta_k = (s_1^k, s_2^k, \ldots, s_{n_k}^k)$ is described by a feature vector in $\mathbb{R}^g$. This means that the representation X_k of a sequence η_k is in turn a sequence of feature vectors, or alternatively, a tensor $X_k \in \mathbb{R}^{n_k \times g}$, where the $t \in \{1, \ldots, n_k\}$ element describes the operation s_t^k. The complete list of g features describing an operation is given in Section 3.4.

Our oracle is composed of two blocks: the *first block* takes in the representation of a sequence X_k and creates a *sequence embedding*; the *second block* uses this embedding to output the probability y_k of the sequence. The entire architecture is depicted in Figure 4.

The first block is realized with two layers of the Gated Recurrent Unit (GRU) [87]. A GRU is a type of Recurrent Neural Network [33] that uses a "memory structure" to let information from prior inputs influence the current output. This "memory structure" needs to be initialized to some initial state, and is updated at each step of the sequence by using the current input and the state through a gating mechanism.

Our oracle warms start the initial states with X_k, but without considering the order. Specifically, the initial state of each GRU layer is created by first projecting the feature vectors describing operations in a latent space $\mathbb{R}^d$ with a hidden layer ($H_0 \in \mathbb{R}^{g \times d}$ and $H_1 \in \mathbb{R}^{g \times d}$ in Figure 4), and then by taking the mean along each of the d dimensions. This allows modeling the concept of a

JSP machine directly in the architecture.

After this initialization starts the creation of the sequence embedding by considering the order of the sequence. As depicted in Figure 4, the first GRU layer receives in input at each step $t = (1, \dots, n_k)$ the feature vector of the operation s_t^k and produces in output the state h_t^0. Whereas the second GRU layer receives in input at the step t the state h_t^0 and produces in output h_t^1. The final sequence embedding is the concatenation of the last states, $h_{n_k}^0$ and $h_{n_k}^1$, and is therefore a vector in $\mathbb{R}^{2d}$.

The second block is realized with a Feedforward Neural Network (FNN) [33] composed of 3 hidden layers of decreasing size. This block takes in input the sequence embedding and produces in output the probability y_k.

3.3 Tabu Search

Since Tabu Search empirically demonstrated to be the best meta-heuristic for solving the JSP [69], we evaluate the advantages of our novel learning task in this algorithm. To this end, we design two versions of the TS: sTS is a simple TS inspired by the works reviewed in Section 2.2.3, while oTS is identical to sTS but uses the oracle. We borrow part of the structure of sTS and oTS from the TS proposed in [7]. Since our algorithms are almost identical, they differ only in the procedure to select the next solution, we first describe the structure of sTS and afterward the modification to the searching procedure.

The key components of sTS are: (i) the generator of the initial solution, (ii) the neighborhood structure, (iii) the tabu list to prevent revisiting recent solutions, (iv) the neighborhood searching procedure to select the next solution, and (v) the restart list to intensify promising regions of the solution space.

sTS begins by generating a random solution that constitutes both the starting point of the exploration and the initial best solution. This solution is generated with a random PDR that gives priority to jobs by sampling from a uniform distribution. We decided to use a random starting point to test the capability of our algorithms to converge to global optima in different runs of the same instance. This allows a better comparison between the algorithms.

After this initialization, sTS enters the cyclic phase where the following steps are repeated:

Step 1: *Create the neighborhood* of the current solution.

Step 2: Select the new current solution through the *neighborhood searching procedure*.

Step 3: *Update the best solution* if the new solution improves the best one.

Step 4: Save a restart point in the restart list if the *region is promising*.

Step 5: Go to *Step* 1: if the *iteration condition* is met.

Step 6: Restart from the latest promising region and go to *Step* 1: if the *restart condition* is met.

At each iteration, the algorithm selects from the N1 neighborhood [6] the solution with minimum makespan that is not forbidden by the tabu list (*Step* 2:). Once sTS finds a solution improving the best one (*Step* 3:), it records this point in the restart list (*Step* 4:). Based on [7, 74], a *promising region* of the JSP solution space is a point in which there is an update of the best solution, and such regions must be intensified by trying to explore the entire neighborhood.

This cyclic exploration is repeated until a maximum number of non-improving iterations is reached (*iteration condition* of *Step* 5:), where a non-improving iteration is an iteration that does not improve the best solution. If the iteration condition is not met, the algorithm tries to resume the exploration from the last promising region inserted in the restart list. The *restart condition* of *Step* 6: simply checks that the restart list is not empty. If this condition is not met, the algorithm stops. The pseudo-code of the neighborhood searching procedure, the tabu list, and the restart list can be found in [7]

oTS is identical to sTS, but it uses the oracle to further reduce the N1 neighborhood by excluding solutions that lower the quality of machine permutations. This aligns with the discussion of Section 3.1. Our oracle predicts the likelihood that a sequence (a permutation on some machine $i \in M$) has of belonging to an optimal solution. In N1, a neighbor solution π_n differs from the current solution π_c in only one permutation on a machine. Therefore, we use the oracle to remove all the neighbor solutions that introduce a sequence with a lower likelihood of belonging to an optimal solution. More in detail, if the permutation of π_n on machine i has a higher likelihood of belonging to an optimal solution than π_c, we accept this solution in the neighborhood, in the opposite case, we remove π_n from the neighborhood. The searching procedure for selecting a new solution from this reduced neighborhood remains the same of sTS, which in turn is the same of [7]. There might be situations in which all the neighbor solutions are removed, in these cases, we undo the reduction and use the normal N1 neighborhood. Finally, this reduction is applied only for the first quarter of the maximum number of non-improving iterations (*Step* 1-5), and in the same way after every restart.

3.4 The Supervised Dataset

For training our oracle, we created a dataset of sequences from a set of 200 JSP instances with 8 jobs ($n_i = 8, \forall i \in M$) and 8 machines ($m_j = 8, \forall j \in J$). The set of instances has been generated following the guidelines of [88].

Then, for each instance, we generated 136 sequences for each machine, and we computed the quality of these sequences with the MILP introduced in Section 3.1. This results for a single instance $q \in \{1, \ldots, 200\}$ in a total of 1088 observations of the form (X_k^q, y_k^q), where $X_k^q \in \mathbb{R}^{n_i \times g}$ is the representation of a machine sequence η_k, and y_k^q is its quality. To ease the notation, in the remainder, we omit the index of the instance q; nonetheless, remember that each observation of our dataset refers to one and only one instance.

The 136 sequences for each machine have been generated as follows:

Algorithm 1 Generate $s \in \mathbb{N}$ random sequences from $(s_1^k, \ldots, s_{n_k}^k)$

function SEQUENCEGENERATOR($(s_1^k, \ldots, s_{n_k}^k), s$)
  $seq \leftarrow$ Generate s empty sequences
  $w \leftarrow (s_1^k, \ldots, s_{n_k}^k)$
  for all $pos \in \{1, \ldots, n_k\}$ **do**
    $idx = 0$
    for all $t \in \{0, \ldots, s-1\}$ **do**
      while w_{idx} in seq_t **do**
        $idx = (idx + 1) \bmod |w|$
      end while
      $seq_t \leftarrow seq_t \oplus w_{idx}$
      $idx = (idx + 1) \bmod |w|$
    end for
    $w \leftarrow w \oplus (s_1^k, \ldots, s_{n_k}^k)$ ▷ Increase w's period.
    $w \leftarrow \text{shuffle}(w)$
  end for
  return seq
end function

- 128 *random sequences* by trying to place each operation in all positions of a machine.
- 1 *optimal sequence* taken from the optimal solution of the instance.
- 7 *suboptimal sequences* obtained from the optimal sequence by swapping consecutive operations (we did not swap the first and last operations).

The rationale behind these different sequences is that we tried to uniformly sample the characteristics of a machine in an instance. The 128 random sequences should reflect the "unbiased" impact of the machine on the instance and they are generated with Algorithm 1, where $\oplus$ indicates that an item is appended to a partial sequence. Note that the `SequenceGenerator` procedure may generate repeated sequences. In such cases, we removed the repeated sequences and applied the procedure again to ensure that s different sequences were generated. The optimal sequence is introduced to model the optimality for a machine, and the suboptimal sequences are used to model the neighborhood of an optimal sequence, and hopefully the *Big Valley* phenomenon [74].

Regarding the representation X_k of a sequence η_k, we defined a set of 18 features to describe operations. Our set of features has been constructed by selecting some of the best features from [89] and from the graph theory. The features selected from [89] describe characteristics of single operations and jobs, some examples are: the processing time of operations and the mean processing time of jobs. The graph theory features are extracted from the disjunctive graph and they express relationships among operations, some examples are: the eigenvector centrality and the closeness centrality. These features depend only

on information about the instance, therefore, in our experiments, we computed the feature vector for each operation once and we dynamically concatenated the feature vectors in the order given by η_k to form X_k ($X_k \in \mathbb{R}^{8\times 18}$ in this work). We report in Table 1 the complete set of features.

3.5 Results

For our experimental evaluation, we configured the oracle of Section 3.2 as outlined in Table 2. We recall that the objective of our oracle is to predict from the representation X_k of an input sequence η_k its quality y_k, i.e., the probability of η_k of being in an optimal solution of the instance. Due to the nature of $y_k \in [0, 1]$, we trained our oracle to approximate the distribution of y_k in our dataset by using the Kullback–Leibler Divergence as the loss function. Using this loss allows to train the model without transforming the problem into a binary classification, and this brings several advantages: (i) our labels y_k have a larger semantic compared to binary ones, giving more freedom in the application of the oracle; (ii) it is not clear which threshold should be set on the continuous labels y_k to transform them into binary ones; (iii) casting the problem as a binary classification brings imbalance issues [90].

We split 75/25 the dataset of Section 3.4, and use the 75% for training and the remaining for testing. The oracle was trained with the adam optimizer [33], with a batch size of 128, and for a total of 100 epochs divided as follows:

1. 40 epochs with learning rate 0.005.
2. 30 epochs with learning rate 0.002.
3. 20 epochs with learning rate 0.001.
4. 10 epochs with learning rate 0.0005.

The training of the oracle was done in Python and was successively ported by using the tracing functionality of PyTorch [91]. Both sTS and oTS have been written in C++, compiled with g++ 9.3.0, and executed on an Ubuntu machine equipped with an Intel Core i9-11900K and an NVIDIA GeForce RTX 3090. Finally, the oracle was integrated into the oTS with the LibTorch library and run on the GPU.

3.5.1 Oracle performance

We start the evaluation by showing the performance of the oracle on two different aspects: (i) we quantify the error in the predictions by measuring how much they differ from labels, (ii) we quantify the performance of the oracle in a binary classification problem. The results of this section refer to a test set composed of 54400 sequences (25% of the dataset) randomly selected by ensuring that the test distribution is similar to the one of the entire dataset, see Figure 5.

Table 1: The set of features describing an operation $i \in O$ and its relations with the other operations in the JSP instance.

ID	Name	Equation	Description
f_0	Processing time	$\frac{\tau_i}{\max_{t \in O} \tau_t}$	The processing time of operation i normalized by the maximum processing time in the instance.
f_1	Job completion	$\frac{\sum_{k=l_j}^{i} \tau_k}{\sum_{t \in O_j} \tau_t}$	The completion of job j when its operation i is scheduled.
f_2	Job mean	$\frac{1}{\text{avg} * m_j} \sum_{k \in O_j} \tau_k$	Mean processing time of job j normalized by the mean processing time of the instance (avg in the equation).
f_3	Job median	$\frac{\text{median}(\tau_{l_j}, \ldots, \tau_{l_j+m_j})}{\text{avg}}$	Median processing time of job j scaled by the mean processing time of the instance (avg in the equation).
f_4	Job std-mean	$\frac{\text{std}(\tau_{l_j}, \ldots, \tau_{l_j+m_j})\, m_j}{\sum_{i \in O_j} \tau_i}$	The standard deviation of the processing time in job j normalized by the mean processing time of the job.
f_5	Job std-median	$\frac{\text{std}(\tau_{l_j}, \ldots, \tau_{l_j+m_j})}{\text{median}(\tau_{l_j}, \ldots, \tau_{l_j+m_j})}$	The standard deviation of the processing time in job j normalized by the median processing time of the job.
f_6	Job min	$\frac{\min_{i \in O_j} \tau_i}{\max_{k \in O} \tau_k}$	The minimum processing time of job j normalized by the maximum processing time in the instance.
f_7	Job max	$\frac{\max_{i \in O_j} \tau_i}{\max_{k \in O} \tau_k}$	The maximum processing time of job j normalized by the maximum processing time in the instance.
f_8	Source shortest distance	$d^*(\text{src}, i)$	The shortest weighted distance in the graph from the source (dummy) node to operation i.
f_9	Destination shortest distance	$d^*(i, \text{dst})$	The shortest weighted distance in the graph from operation i to the destination (dummy) node.
f_{10}	Eigenvector Centrality	$Ax = x\lambda$	The eigenvector centrality of an operation i is the element of the eigenvector x associated with the largest eigenvalue λ that corresponds to i. A high eigenvector centrality means that an operation connects to other operations having high centrality. A is the adjacency matrix of the graph.
f_{11}	Weighted Eigenvector Centrality	$A^*x = x\lambda$	The same as the eigenvector centrality, but it uses the weighted adjacency matrix A^* where arcs take the weight of the source node.
f_{12}	Closeness Centrality	$\frac{\lvert V \rvert - 1}{\sum_{k \in \mathcal{N}(i)} d(k,i)}$	The normalized closeness centrality measures the shortest non-weighted distance from the nodes than can reach i, scaled by the number of nodes in the graph. $\mathcal{N}(i)$ is the set of nodes that can reach i, $d(k,i)$ is the number of arcs on the shortest path from k to i, and $\lvert V \rvert$ is the number of nodes.
f_{13}	Weighted Closeness Centrality	$\frac{\lvert V \rvert - 1}{\sum_{k \in \mathcal{N}(i)} d^*(k,i)}$	The same as the closeness centrality, but it uses the weighted shortest path $d^*(k,i)$.
f_{14}	Betweenness Centrality	$\sum_{v,w \in V} \frac{\Gamma_{v \to w}(i)}{\Gamma_{v \to w}}$	The betweenness centrality is the fraction of all-pairs shortest paths that pass through operation i. This measure indicates which operations are "bridges" between others in a graph. $\Gamma_{v \to w}$ is the number of non-weighted shortest paths from v to w, and $\Gamma_{v \to w}(i)$ is the number of such shortest paths through i.
f_{15}	Weighted Betweenness Centrality	$\sum_{v,w \in V} \frac{\Gamma^*_{v \to w}(i)}{\Gamma^*_{v \to w}}$	The same as the betweenness centrality, but it uses the weighted shortest path for computing the number of paths $\Gamma^*_{v \to w}$.
f_{16}	Page Rank	A	The Page Rank.
f_{17}	Weighted Page Rank	A^*	The weighted Page Rank.

Table 2: Hyper-parameters of the oracle presented in Section 3.2.

GRU 0		GRU 1		FNN	
Hyper-parameter	Value	Hyper-parameter	Value	Hyper-parameter	Value
hidden size	32	hidden size	32	H_2 size	32
bidirectional	False	bidirectional	False	H_2 activation	tanh
dropout	0.3	H_1 size	32	H_3 size	16
H_0 size	32	H_1 activation	tanh	H_3 activation	tanh
H_0 activation	tanh	H_1 dropout	0.3	H_4 size	2
H_0 dropout	0.3				

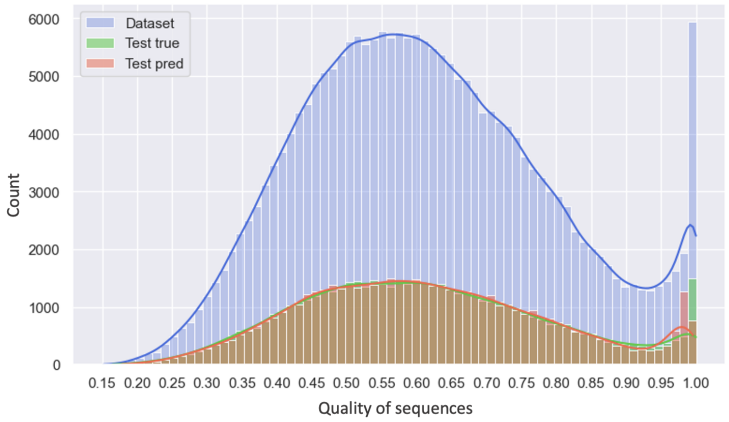

Figure 5: The discretized distributions of the dataset (blue), test set (green), and oracle predictions (red).

To quantify the errors of the oracle, we compare its predictions against the labels of the test set by defining the Within Tolerance Accuracy (WTA) in Equation 11:

$$\text{WTA}(tol) = \frac{1}{T}\sum_{k=0}^{T}\mathbb{I}(|y_k - \hat{y_k}| < tol) \tag{11}$$

where $\hat{y_k}$ is the predicted quality of a sequence η_k, y_k is the true quality, tol is the error tolerance, $\mathbb{I}()$ is the indicator function (it returns 1 when the difference is within the tolerance), and T is the dimension of the test set.

As it is clear from the distributions in Figure 5, the predictions of the oracle approximate well the distribution of the test set, with some mistakes in the region around the quality 0.90. This is even more clear from Table 3 which quantifies the errors between true and predicted values in different portions of the test distribution. This table divides the sequences into intervals based on the true quality. The first column points out the quality intervals, the second column gives the number of occurrences in each interval, the third and fourth columns give some statistics about the absolute error between labels and predictions,

Table 3: The errors and WTA for different quality intervals.

Quality	Num	Abs error		WTA(.05)	WTA(.07)
		avg	max	(%)	(%)
$[0.0, 0.3)$	1117	0.012	0.078	98.0	99.7
$[0.3, 0.4)$	4756	0.015	0.111	95.8	99.3
$[0.4, 0.5)$	9710	0.016	0.158	94.2	98.6
$[0.5, 0.6)$	11775	0.018	0.150	91.7	97.8
$[0.6, 0.7)$	10808	0.020	0.171	89.2	97.1
$[0.7, 0.8)$	7702	0.020	0.158	87.3	96.7
$[0.8, 0.9)$	4343	0.021	0.152	85.9	96.2
$[0.9, 1.0]$	4189	0.018	0.123	91.3	97.9
Test set WTA				91.0	97.7

and the last two columns give the WTA for different tolerances. The last row gives the WTA for the entire test set.

Note how the WTA is almost perfect for a tolerance of 0.07 and still very good for a tighter tolerance of 0.05. As noted above, we can appreciate an increment in the errors in the interval $[0.7, 0.9)$. We believe that this increment is jointly caused by the lower number of training sequences in this interval and by the fact that these sequences are more difficult to discriminate from optimal ones because they are mostly suboptimal, i.e., they differ from optima only in one consecutive pair of operations.

To better understand the quality of the oracle, we also report in Table 4 its performance in a binary classification task. In this evaluation, the true quality y_k is transformed into binary labels by setting a threshold and marking with a 1 (positive) all the sequences having a quality higher than the threshold, and with a 0 (negative) all the remaining sequences. The class predicted by the oracle is given by the *argmax* function. We report the results for 5 different thresholds, each producing a binary test set with a different imbalance ratio. Despite these different imbalance ratios, the performance of the oracle on standard imbalanced metrics [90] remains good in all cases. This is possible because we trained the model to match the quality of the training sequences.

With these evaluations, we want to stress how the oracle can be effectively used either for predicting or classifying sequences, allowing great flexibility in its usage within our TS and potentially in other approximation methods. In addition, the results of this section suggest that with our dataset, and hence with the methodology of Section 3.1, it is possible to learn which sequences are likely to be in an optimal solution.

Table 4: The results on 5 binary classification problems obtained by setting 5 different thresholds on the labels of the test set.

Threshold	Num positive	Num negative	Imbalance ratio	Accuracy (%)	Balanced accuracy (%)	Precision (%)	Recall (%)
0.5	38817	15583	0.4	95.5	94.3	96.6	97.0
0.6	27042	27358	1.01	94.7	94.7	94.6	94.8
0.7	16234	38166	2.35	95.4	94.5	92.3	92.2
0.8	8532	45868	5.38	97.1	94.0	91.9	89.4
0.9	4189	50211	11.99	98.6	94.4	92.2	89.4

Table 5: The results of the two TS algorithms for different configurations of parameters.

	Parameters		Num opt		Avg gap (%)		Worse		Better		Avg time (ms)	
ID	Max iter	Rest.	sTS	oTS	sTS	oTS	Num	Avg diff (%)	Num	Avg diff (%)	sTS	oTS
0	500		335	465	2.24	2.12	277	1.56	408	1.93	29	142
1	1000		449	620	1.81	1.77	192	1.32	372	1.56	41	210
2	1500	0	511	673	1.66	1.41	167	1.06	341	1.55	56	318
3	2000		559	741	1.55	1.29	135	1.11	320	1.49	72	409
4	2500		593	763	1.51	1.34	116	1.10	293	1.46	86	473
5		0	382	548	1.99	1.75	223	1.22	406	1.75	33	153
6	700	1	741	849	1.02	0.92	80	0.93	195	1.02	138	526
7		2	807	892	0.89	0.92	65	0.97	150	0.89	199	690
8		0	409	572	1.89	1.71	211	1.28	395	1.65	35	165
9	800	1	754	867	1.01	0.84	71	0.79	185	1.04	181	592
10		2	818	905	0.89	0.85	58	0.80	140	0.91	245	795
11		0	425	594	1.84	1.69	218	1.24	377	1.70	38	191
12	900	1	766	879	1.00	0.97	66	0.96	173	1.04	204	714
13		2	833	918	0.88	0.82	48	0.93	128	0.97	263	961

3.5.2 Tabu Search performance

We proceed by analyzing the impact of the proposed learning task on the Tabu Search meta-heuristic. To this end, we compare the results of the TS described in Section 3.3 with (oTS) and without (sTS) the oracle. This comparison is done on the 200 instances used to create our dataset, where for each instance we repeated the execution of the algorithms 5 times, from the same 5 initial solutions (this is done by seeding the random PDR with 5 different seeds). The results of the algorithms are compared in terms of the number of optimal solutions, the average optimality gap of suboptimal solutions (gap $= (C_{max}/C^{opt}_{max}) - 1$), and the average execution time. Note that comparing the results of the algorithms on the same instances of our dataset is fair since the sequences visited by sTS and oTS are independent from those used to train the oracle.

In Table 5, we report the results of the algorithms for different configurations of parameters. In these configurations, we omit the length of the tabu list that is always set to 10. The first column of the table assigns an identifier to every

configuration. The second and third column specifies respectively the maximum number of non-improving iterations and the length of the restart list. The "*Num opt*" and the "*Avg gap*" columns compare the number of optimal solutions and the average optimality gap of each algorithm. Whereas the "*Worse*" ("*Better*") columns compare respectively the number of solutions and the average scaled difference (diff = $(C^{oTS}_{max} - C^{sTS}_{max})/C^{opt}_{max}$) in which oTS worsens (improves) with respects to sTS. The last two columns give the average execution times.

First, we underline that oTS finds a higher number of optimal solutions than sTS regardless of the parameter configurations. This is important for empirically confirming that the proposed learning task, our methodology, and the learning model indeed enhance the performance of the TS.

This increment in performance is also supported by the lower average optimality gaps obtained by oTS in suboptimal solutions ("*Avg gap*" columns). For all the tested configurations, we only see one case, row with ID 7, in which oTS does slightly worse than sTS in terms of optimality gap. However, note how in this case the overall performance of both the algorithms is almost perfect, and how oTS is still able to find a larger number of optimal solutions.

Regarding the "*Worse*" and the "*Better*" columns, we highlight how the number of solutions in which oTS does better than sTS is almost twice the number of solutions in which it does worse. Furthermore, it is worth noting that with more restarts, the average difference in the gaps of the algorithms tend to decrease as they both converge towards optimality, rendering the oracle suggestions less advantageous.

Finally, as it is clear from the average times, using a deep learning model will likely increase the running time. This trend has already been observed in [13], where the execution of a DRL proposal takes 2x up to 5x the time of traditional PDRs. A similar increment is also observed in [78]. In line with these works, we observe a comparable increment between sTS and oTS. However, our algorithms have been written by keeping the implementation as simple as possible. Therefore, there is space for engineering the code and producing better average execution times, especially in the case of the oTS. For instance, it is possible to reduce the oracle calls by batching or keeping a memory of past predictions, and it is possible to reduce the execution time of the oracle by using faster architectures like Transformers [92] and Convolutional Network [33].

Concluding, this comparative analysis shows that it is possible to find better solutions by using the quality predictions in a TS as described in Section 3.3. This empirically highlights how the proposed learning task seems to be valuable in the context of the JSP.

3.5.3 Comparison with Reinforcement Learning

Lastly, we compare oTS with RL proposals reviewed in Section 2.2.4. Our goal is to prove the superiority of meta-heuristics enhanced with machine learning and stress the importance of further investigating these hybrid approaches.

For this comparison, we selected standard benchmark instances from works discussed in Section 2.2.4. Specifically, we selected the instances Orb01-09 [93]

Table 6: The comparison between oTS and the RL approaches on benchmark instances.

	Instance	OPT	SPT	[78]	[13]	[83]	oTS-1	oTS-2
10×10	Orb01	1059	1478 (39.6%)	1211 (14.4%)	-	-	1106 (4.4%)	1106 (4.4%)
	Orb02	888	1175 (32.3%)	1002 (12.8%)	-	-	902 (1.6%)	902 (1.6%)
	Orb03	1005	1179 (17.3%)	1150 (14.4%)	-	-	1048 (4.3%)	1044 (3.9%)
	Orb04	1005	1236 (23.0%)	1132 (12.6%)	-	-	1032 (2.7%)	1032 (2.7%)
	Orb05	887	1152 (29.9%)	1045 (17.8%)	-	-	902 (1.7%)	896 (1.0%)
	Orb06	1010	1190 (17.8%)	1106 (9.5%)	-	-	1028 (1.8%)	1028 (1.8%)
	Orb07	397	504 (27.0%)	460 (15.9%)	-	-	397 (0.0%)	397 (0.0%)
	Orb08	899	1170 (30.1%)	1022 (13.7%)	-	-	911 (1.3%)	911 (1.3%)
	Orb09	934	1262 (35.1%)	1082 (15.8%)	-	-	961 (2.9%)	955 (2.2%)
15×15	Ta01	1231	1872 (52.1%)	-	1443 (17.2%)	1320 (7.23%)	1281 (4.1%)	1281 (4.1%)
	Ta02	1244	1709 (37.4%)	-	1544 (24.1%)	1311 (5.39%)	1283 (3.1%)	1283 (3.1%)
	Ta03	1218	2009 (64.9%)	-	1440 (18.2%)	1318 (8.21%)	1292 (6.1%)	1292 (6.1%)
	Ta04	1175	1825 (55.3%)	-	1637 (39.3%)	1287 (9.53%)	1248 (6.2%)	1248 (6.2%)
	Ta05	1224	2044 (67.0%)	-	1619 (32.3%)	1323 (8.09%)	1280 (4.6%)	1280 (4.6%)
	Ta06	1238	1771 (43.1%)	-	1601 (29.3%)	1311 (5.89%)	1272 (2.7%)	1260 (1.8%)
	Ta07	1227	2016 (64.3%)	-	1568 (27.8%)	1270 (3.50%)	1250 (1.9%)	1247 (1.6%)
	Ta08	1217	1654 (35.9%)	-	1468 (20.6%)	1305 (7.23%)	1240 (1.9%)	1240 (1.9%)
	Ta09	1274	1962 (54.0%)	-	1627 (27.7%)	1461 (14.68%)	1307 (2.6%)	1307 (2.6%)
	Ta10	1241	2164 (74.4%)	-	1527 (23.0%)	1334 (7.49%)	1290 (3.9%)	1290 (3.9%)

and the instances Ta01-10 [88]. In Table 6, we report for each instance its name, the optimal makespan, and the results in terms of makespan and optimality gap (in round brackets) for the Shortest Processing Time (SPT), the proposal in [78], the proposal in [13], that of [83], and oTS. Based on Table 5, we decided to use 2 parameter configurations for oTS: 2 restarts and 700 iterations for oTS-1, and 2 restarts and 800 iterations for oTS-2.

From Table 6, it is immediately clear that oTS outperforms the DRL proposals. This is also true if we qualitatively compare the results of oTS with those reported in [79]. By looking at the average percentage gap reported for Orb01-10 and Ta01-80, we can see that the gap of this other DRL proposal is around 20%, ten times the gap obtained by oTS.

This comparison demonstrates that meta-heuristics enhanced with machine learning guarantee to find better solutions. We believe that further research in hybrid approaches as our may give life to simpler and better meta-heuristics capable of producing near-optimal solutions in a shorter amount of time.

3.6 Final Remarks

This study showcases the feasibility of leveraging supervised methodologies to acquire informative insights about CO problems, thereby enhancing optimization techniques like meta-heuristics. Furthermore, we emphasize that ML approaches can effectively approximate assumptions derived from theoretical proofs about optimization methods, which might not be of practical use otherwise. This, in turn, enables the implementation of enhanced methods where the

ML component generally has only a limited negative impact on the method.

Finally, we underscore the significance of a methodology for learning the quality of machine permutations. Such an approach holds valuable implications and can be seamlessly extended to a diverse array of algorithms and scheduling problems.

4 Self-Labeling the Job Shop Problem

Over the years, researchers have developed a variety of techniques to tackle the JSP, including exact methods [4], heuristics [3], and meta-heuristics [6, 7]. Although many techniques exist, they still have limitations, and none is appropriate for every circumstance. Some related limitations are: (i) only few of the best techniques can *scale* well on large instances; (ii) most techniques relying on heuristics or pre-defined rules, such as PDRs, fail to be *flexible* when exposed to different instances; (iii) incorporating *domain knowledge* about jobs and machines into existing techniques can be challenging and time-consuming.

To address some of these challenges, we investigate again Machine Learning paradigms, which have provided several technological breakthroughs in the last decades. Differently from Section 3, where we employed ML to enhance meta-heuristics, we now focus on solving the JSP by resorting only to ML models in an end-to-end fashion. By learning general and flexible patterns, ML models can rapidly and effectively solve large and complex instances than part of traditional JSP algorithms. Additionally, incorporating domain knowledge while training models is as easy as personalizing the training set, and this may lead to superior solutions in real-world applications.

Despite these premises, how to effectively and efficiently apply ML to solve the JSP remains an open question. Recently, significant research efforts focus on applying Reinforcement Learning to the JSP (see Section 2.2.4). The driving interest in RL approaches is that they allow learning neural solvers through trial-and-error schemes, thus avoiding expensive supervision. However, training RL agents is a complex optimization task, has reproducibility issues [16], and is computationally expensive [13]. Therefore, in this work, we specifically focus on designing an efficient supervised methodology.

In traditional *supervised learning*, a model is trained to associate each example with a single class through the aid of pre-defined labels. A natural idea is to apply a similar approach whereby a model learns a mapping between a JSP instance and its optimal solution. However, as supervised learning is known to require several thousands of annotations [18], this approach is particularly problematic in combinatorial problems, where annotations in the form of optimal solutions are generally produced with expensive exact methods.

As explained in Section 2.1.2, *Semi- and Self-Supervised learning* [19] are recently gaining popularity in many ML fields due to their ability to learn from fewer labeled data. This allows reducing labeling costs and also improves generalization [18]. However, little to no application of these paradigms can be found in the JSP and the combinatorial optimization literature [20].

We thus focus on the design of a supervised training procedure relying only on model-generated solutions and not requiring expensive (near-)optimal solutions. Our proposal is based on the following two assumptions: i) we suppose to be able to generate *multiple solutions* for an instance, a common characteristic of generative ML models like the Pointer Network [94]; and ii) we suppose it is possible to *discriminate solutions based on the problem objective*. When these assumptions are met, we train a model by generating multiple solutions and using the best one according to the problem objective as a pseudo-label [38]. This procedure borrows from semi-supervised learning the idea of pseudo-labeling but does not resort to additional annotations as in self-supervised learning. Hence, we refer to it as a *Self-Labeling training strategy*.

Similar to constructive heuristics described in Section 2.2.2, we cast the generation of JSP solutions as a sequence of decisions, where at each decision one operation is selected to be added to the solution under construction. This is achieved with a popular architecture known as *Pointer Network* (PN) for dealing with sequences of decisions. We refer the reader to Section 2.1.3 for an in-depth treatment of the PN. We train our PN on a set of 30 000 instances by generating multiple parallel solutions and using the one with the *minimum makespan* to update the model. Quite surprisingly, this simple training strategy produces models outperforming existing state-of-the-art RL proposals for the JSP on two popular benchmark sets.[0]

Note:

An extended version of this work is currently under revision at a top ML conference and was presented at:
Andrea Corsini, and Mauro Dell'Amico, "*Using Self-Supervised Learning to Solve the Job Shop Scheduling Problem*". International Conference on Optimization and Decision Science, 2023. Ischia, IT.

4.1 Proposed Pointer Neural Network

Since we tackle the JSP as a sequence of decisions (see Section 2.2.2), we propose a PN whose goal is to select the right job at each decision step t. Formally, our PN learns a function $f_\theta(\cdot)$, parametrized by θ, that estimates the probability $p_\theta(\pi|G)$ of a solution π of being of high-quality as a product of probabilities:

$$p_\theta(\pi|G) = \prod_{t=1}^{o} f_\theta(j \mid \pi_t, G), \tag{12}$$

[0]The code is available at: https://github.com/AndreaCorsini1/SelfLabelingJobShop

where $f_\theta(j\,|\pi_t, G)$ gives the probability of selecting $j \in J$ conditioned on the instance G and the partial solution π_t at step t. By learning f_θ, it is possible to construct solutions autoregressively with a high chance of being of high-quality.

Our PN works by taking in input JSP instances represented as a disjunctive graph G, and produces a single deterministic or multiple randomized solutions depending on the adopted construction strategy. Following the other ML approaches for the JSP reviewed in Section 2.2.4, we augment the disjunctive graph associated to an instance with additional features for each vertex. We provide the complete list such features x_i describing an operation i in Table 7.

As described in Section 2.1.3, the PN comprises an encoder and a decoder. The *encoder* of our PN creates with a single forward computation of G an embedded representation for all the operations in O. Whereas the *decoder* uses the embeddings of operations and the partial solution π_t to produce a probability for each job of being selected at step t. Recall that after selecting a job j, its ready operation $o(t,\, j)$ is scheduled in π_t to produce the partial solution for step $t+1$. In the following, we present the details of both the encoder and decoder.

Encoder.

Its role is to encode meaningful patterns within an instance into the embedded representation of operations. The encoder can be embodied by any architecture, like a Feedforward Neural Network (FNN), that transforms the feature of operations $x_i \in \mathbb{R}^{15}$ into embeddings $e_i \in \mathbb{R}^h$, without necessarily accounting for information contained in the graph G. As in related works [13, 79], we propose to additionally encode the relationships among operations present in the disjunctive graph. Therefore, we equip the encoder with graph neural networks [55], which enables the embeddings to also incorporate information related to the topological structure of G. In our encoder, we stack two layers of Graph Attention Network [95] (GAT) as follows:

$$e_i = [x_i \,||\, \sigma(\mathrm{GAT}_2(\,[x_i \,||\, \sigma(\mathrm{GAT}_1(x_i,\, G))],\, G)], \tag{13}$$

where σ is the ReLU non-linearity and $||$ stands for the concatenation operation.

Decoder.

Its role is to produce at any step t the probability of selecting each job with two logically distinct components:

- *Memory Network*: generates a state $s_j \in \mathbb{R}^d$ for each job $j \in J$ from the partial solution π_t. This is achieved by first extracting from π_t a context vector c_j for every job j, which contains eleven hand-crafted features providing useful cues about the job in the partial solution. We refer to Table 8 for the definition of such features. Then, these vectors are fed into a Multi-Head Attention (MHA) layer [57] followed by a non-linear projection to produce the jobs' states:

$$s_j = \sigma([c_j\, W_1 + \underset{b \in J}{\mathrm{MHA}}(c_b\, W_1)]\, W_2), \tag{14}$$

Table 7: The features $x_i \in \mathbb{R}^{15}$ associated with a vertex $i \in V$ describing information about operation i within the instance.

ID	Description
1	The processing time τ_i of the operation.
2	The completion of job j up to operation i: $\sum_{b=l_j}^{i} \tau_b \,/\, \sum_{b \in O_j} \tau_b$.
3	The remainder of job j from operation i: $\sum_{b=i+1}^{l_j+m_j} \tau_b \,/\, \sum_{b \in O_j} \tau_b$.
4-6	The 1^{st}, 2^{nd}, and 3^{rd} quartiles among processing times of operations on job j.
7-9	The 1^{st}, 2^{nd}, and 3^{rd} quartile among processing times of operations on machine μ_i.
10-12	The difference between τ_i and the 1^{st}, 2^{nd}, and 3^{rd} quartiles among processing times of operations in job j.
13-15	The difference between τ_i and the 1^{st}, 2^{nd}, and 3^{rd} quartiles among processing times of operations on machine μ_i.

where W_1 and W_2 are two projection matrices, and σ is the ReLU function. Note that we use the MHA to consider the context of all jobs when producing the state for a specific one, similarly to [58].

- *Classifier Network*: outputs the probability p_j of selecting a job j by combining the embedding $e_{o(t,\,j)}$ of its ready operation and the state s_j produced by the memory network. To achieve this, we first concatenate the embeddings $e_{o(t,\,j)}$ with the states s_j and apply an FNN:

$$z_j = \text{FNN}([e_{o(t,\,j)} \,||\, s_j]). \tag{15}$$

Then, these scores $z_j \in \mathbb{R}$ are transformed into probabilities with a Softmax function: $p_j = e^{z_j} \,/\, \sum_{b \in J} e^{z_b}$. Finally, the decision of which job to select at t is made with a sampling strategy, as explained next.

Sampling solutions.

Traditionally, the PN generates a single greedy solution by scheduling at each step t the operation of the job j with the highest probability p_j [13, 58]. To generate solutions, we instead employ a probabilistic approach for deciding the job selected at any step. Specifically, we randomly sample a job j with a probability proportional to p_j, which is produced at step t by our decoder. We also prevent the selection of completed jobs by setting their probabilities to zero before sampling, i.e., we set $p_j = 0$ for completed jobs j. Note how this probabilistic selection is *autoregressive*, meaning that sampling a job depends

Table 8: The features of a context vector that describes the status of a job j within a partial solution π_t. Recall that $o(t, j)$ is the ready operation of job j at step t and $o(t, j) - 1$ its predecessor, if it does not exist $C_{o(t,\,j)-1}(\pi_t) = 0$.

ID	Description
1	$C_{o(t,\,j)-1}(\pi_t)$ minus the completion time of machine $\mu_{o(t,\,j)}$.
2	$C_{o(t,\,j)-1}(\pi_t)$ divided by the makespan of π_t.
3	$C_{o(t,\,j)-1}(\pi_t)$ minus the average completion time of all jobs.
4	The completion time of machine $\mu_{o(t,\,j)}$ divided by the makespan of the partial solution π_t.
5	The completion time of machine $\mu_{o(t,\,j)}$ minus the average completion time of all machines in π_t.
6-8	The difference between $C_{o(t,\,j)-1}(\pi_t)$ and the 1^{st}, 2^{nd}, and 3^{rd} quartile computed among the completion time of all jobs.
9-11	The difference between the completion time of machine $\mu_{o(t,\,j)}$ and the 1^{st}, 2^{nd}, and 3^{rd} quartile computed among the completion time of all machines.

on p_j which is a function of $e_{o(t,\,j)}$ and s_j resulting from earlier decisions (see Eq. 14 and 15). In literature, there are various strategies for sampling from autoregressive models, including top-k sampling [54], nucleus sampling [54], and random sampling [96]. After preliminary analysis, we found that these strategies exhibit similar effectiveness in our setting. Therefore, we adopt the simplest strategy for training and testing, which is *random sampling*, i.e., sampling a job j with probability p_j at random. Lastly, we use our PN to generate β parallel solutions for an instance. This is achieved by keeping β separate partial solutions and sampling at any step a job for each one independently.

4.2 Self-Labeling Training Strategy

To train generative models, we propose a simple self-supervised strategy that resorts only to the model being trained and does not require optimality information nor the formulation of the Markov Decision Process. Our strategy exploits two aspects: the ability of generative models to construct multiple (parallel) solutions, and the possibility of discriminating solutions based on their objective values, such as the makespan in the JSP. With these two ingredients, we design a procedure that at each iteration generates multiple solutions and uses the best one as a *pseudo-label* [38], see Figure 6 for a graphical illustration.

Specifically, for each training instance G (either randomly created or loaded from a dataset), we generate with the model a set of β different solutions. We

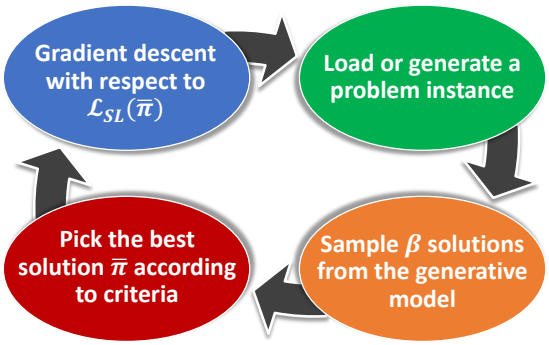

Figure 6: The self-labeling strategy for training generative models as the Pointer Network of Section 4.1.

do that by keeping β partial solutions in parallel and independently sample for each one the next job from the probabilities generated by the decoder. Once the solutions have been generated, we take the one with the minimum makespan $\overline{\pi}$ and use it to compute the Self-Labeling loss ($\mathcal{L}_{\text{SL}}$) by evaluating the *cross-entropy* on all the decision steps as follows:

$$\mathcal{L}_{\text{SL}}(\overline{\pi}) = -\frac{1}{o}\sum_{t=1}^{o} \log f_\theta(y_t \mid \pi_t, G), \tag{16}$$

where o is the number of operations in the instance G and $y_t \in J$ is the index of the job selected at the decision step t while constructing the solution $\overline{\pi}$.

The rationale of this training schema is to collect knowledge about sequences of decisions leading to high-quality solutions and save it in the model parameters θ. This is done by selecting the best-generated solution $\overline{\pi}$ for an instance G and maximizing its likelihood with Eq. 16 as in supervised learning, i.e., by minimizing cross-entropy. Through iterative exposure to different training instances, the model refines its ability to effectively solve the JSP.

Although our strategy was developed independently, attentive readers may find a resemblance with the Cross-Entropy Method (CEM), a stochastic and derivative-free optimization method [97]. Typically applied to optimize parametric models such as Bernoulli and Gaussian mixtures models [98], the CEM relies on a maximum likelihood approach to either estimate a random variable or optimize the objective function of a problem [99]. According to Algorithm 2.2 in [99], the CEM independently samples $N = \beta$ solutions, selects a subset based on the problem's objective ($\hat{\gamma} = S_{(N)}$ in our case), and updates the parameters.

Unlike the CEM, which independently tackles different instances by optimizing a separate model for each, our strategy trains a single model on many instances for learning how to globally solve the JSP. Moreover, we adopt a more complex parametric model rather than mixture models, always select a single solution for updating, and resorting to the gradient descent for updating parameters. It is noteworthy that our strategy also differs from CEM applications

to model-based RL, e.g., [98], as we do not rely on rewards in Eq. 16. Summarizing, our strategy shares the idea of sampling and selecting with the CEM, but operates globally as a fully supervised learning once target solutions are identified. Hence, we named it: *self-labeling training strategy*.

Other self-labeling approaches can be found e.g. in [43, 100], where the former uses K-Means to generate pseudo-labels, and the latter assigns labels to equally partition data with an optimal transportation problem. As highlighted in [100], these approaches may incur in degenerative solutions, i.e., solutions that trivially minimize Eq. 16, such as producing the same solution despite the input instance in our case. We remark that our strategy avoids such solutions by jointly using a probabilistic generation process and the objective value (makespan) of solutions to select the best pseudo-label $\overline{\pi}$.

4.3 Experimental Setup

Dataset and Benchmarks.

To train our Pointer Network, we created a dataset of 30 000 instances following [88]. For generating instances, we arbitrarily selected six instance shapes ($n \times m$): $\{10 \times 10,\ 15 \times 10,\ 15 \times 15,\ 20 \times 10,\ 20 \times 15,\ 20 \times 20\}$, and randomly generate 5000 instances per shape. Although a fixed dataset is not strictly required by our self-labeling training strategy (instances can be generated on the fly), we prefer using it to favor reproducibility and consistently track training statistics such as average loss and performance.

We also adopted two popular benchmark sets to evaluate our model and favor the cross-comparison with other works. The first benchmark is the well-known from Taillard[1] [88], containing 80 instances of medium and large shapes (10 instances per shape). The second one is from Demirkol's work[1] [101], containing 80 instances of medium-large shapes. Although this set is less popular than others, it demonstrated to contain challenging instances [13, 14]

Architecture.

In all our experiments, we configure and train the Pointer Network of Section 4.1 in the same way. Our *encoder* consists of two Graph Attention layers [95], both with 3 attention heads and Leaky slope at 0.15. In GAT_1, we set the size of each head to 64 and concatenate their outputs; while in GAT_2, we increase the head's size to 128 and average their output to produce $e_i \in \mathbb{R}^{143}$ ($h = 15 + 128$). Inside the *decoder's memory network*, the MHA layer follows [57] except it concatenates the output of 3 heads with 64 neurons each, while $W_1 \in \mathbb{R}^{11\times192}$ and $W_2 \in \mathbb{R}^{192\times128}$ use 192 and 128 neurons, respectively. Thus, in Eq. 14, states $s_j \in \mathbb{R}^d$ have size $d = 128$. Regarding the *classifier*, the FNN features a dense layer with 128 neurons activated through the Leaky-ReLU non-linearity (slope = 0.15) and a final linear layer with a single neuron.

Training.

We train this model on our dataset for 50 epochs with the Adam optimizer [33] and save the parameters producing the best results on a hold-out set comprising 100 random instances of each shape. We set the learning rate to 0.0002, and we reduced it by 5 after 5 epochs in which the loss does not decrease below the minimum. Additionally, we fix the number of sampled solutions β in training to 256, and use a batch size of 32, hence accumulating the gradients across 16 instances before updating θ. Training this way roughly takes 140 hours, while with smaller β the time decreases down to 24 hours when $\beta = 32$.

Competitors.

We compare the effectiveness of our PN and training strategy against various types of conventional and ML competitors. As not all the competitors can generate multiple solutions, either because they are not freely available or they do not support randomization, we divide them as follows:

- *Greedy constructives* generate a single solution for any input instance. We consider ML works of Section 2.2.4 tested at least on 20 benchmark instances, and conventional JSP algorithms referenced in such works. This includes the Actor-Critic (L2D) of [13]; the GNN (L2S) of [79]; the Transformer (TRL) of [58]; the Deep Q-Network (DQN) in [80]; and the Curriculum Learning approach (CL) of [14]. We also coded the INSertion Algorithm (INSA) of [7] along with three standard dispatching rules that prioritize jobs based on the Shortest Processing Time (SPT); Most Work Remaining (MWR); and Most Operation Remaining (MOR).

- *Randomized constructives* generate multiple solutions for an instance by introducing a controlled randomization in the selection process, a simple strategy for enhancing conventional and RL algorithms [47, 14]. We consider the three dispatching rules above, randomized by arbitrarily selecting the operation to schedule among the three with higher priority; the randomized results of CL; and those of L2D. Since only greedy results are disclosed for L2D, we used the open-source code and trained model to generate randomized solutions. All these approaches were seeded with 12345, generate $\beta = 128$ solutions, and return the one with minimum makespan.

- *Non-constructive approaches* do not rely on a pure constructive strategy for creating JSP solutions. We include the enhanced metaheuristic (NLS_A) of [102], which is proven to outperform standard JSP metaheuristics; the hybrid CP (hCP) proposal of [85]; and the results of Gurobi 9.5 (MIP) solving the disjunctive JSP formulation of [4] with a time limit of 3600 seconds.

4.4 Results

This section compares the performance of our Self-Labeling Pointer Network (SPN), configured and trained as explained in Section 4.3, with the selected competitors. When contrasted with greedy constructive approaches, our SPN generates a single solution by picking the job with the highest probability. In the other comparisons, we randomly sample 128 (SPN_{128}) and 512 (SPN_{512}) solutions as explained in Section 4.1.

We coded in Python 3.9 and PyTorch 13.1 our SPN, INSA, dispatching rules, and the MILP. Whereas we report results from original papers of all the ML proposals but L2D, for which we used the open-source code and trained models for generating multiple solutions. All our experiments were performed on an Ubuntu 22.04 machine equipped with an Intel Core i9-11900K and an NVIDIA GeForce RTX 3090 having 24GB of memory. Finally, we report performance in terms of the Percentage Gap (PG):

$$\text{PG} = 100 \cdot (C_{alg} \,/\, C_{ub} - 1), \tag{17}$$

where C_{alg} is the makespan produced by an algorithm for a benchmark instance and C_{ub} is either the instance optimal or best-known makespan[1].

4.4.1 Performance on Benchmarks

As some ML algorithms were tested only on certain benchmark instances, we divide this evaluation in two parts. We start by considering the implemented algorithms and ML proposals which were tested on all the instances of the benchmarks. Recall that both the benchmarks comprise 80 instances, 10 instances per shape. Then, we compare all the algorithms on the instances selected in [58, 80].

Table 9 presents the comparison of algorithms tested on all the instances of Taillard's and Demirkol's benchmarks, each arranged in a distinct horizontal section. This table is vertically divided into *Greedy Constructive*, *Randomized Constructive*, and *Non-constructive approaches*, with the results of algorithms categorized accordingly. Competitors not included in the table are discussed in the text as we could not compute their PGs. Each row reports the average PG on a specific instance shape, with the best gap highlighted in bold. The last row (Avg) reports the average gap across all instances, regardless of their shapes.

Table 9 shows that our SPN and CL produce lower gaps than PDRs and INSA in both the greedy and randomized case. Surprisingly, INSA and PDRs outperform L2D. This is likely related to how PDRs were coded in [13], where ours align with those in [14]. Focusing on our SPN, we observe that it consistently archives lower average gaps than all the greedy competitors and, when applied in a randomized manner (SPN_{128}), it outperforms all the randomized constructives. We also underline that our SPN outperforms L2S [79], as it obtains an average PG of 13.37% on Taillard's benchmark while L2S an average of

[1] Available at: https://optimizizer.com/jobshop.php

Table 9: The average PGs of the algorithms on the two benchmark sets. We highlight in **bold** the best performance (lowest gap) on each row. Shapes marked with * are larger than those seen in training by our SPN.

		Greedy Constructives							*Randomized Constructives*							*Non-constructive Approaches*	
		PDRs				RL		Our	PDRs ($\beta = 128$)			RL ($\beta = 128$)		Our			
	Shape	SPT	MWR	MOR	INSA	L2D[13]	CL[14]	SPN	SPT	MWR	MOR	L2D[13]	CL[14]	SPN_{128}	SPN_{512}	NLS_A[102]	MIP
Taillard's benchmark	15×15	26.09	19.09	20.40	14.43	25.96	14.26	13.77	13.47	13.47	12.53	17.11	9.02	7.24	6.52	7.74	**0.07**
	20×15	32.30	23.33	24.86	18.88	30.03	16.52	14.96	18.45	17.17	16.38	23.74	10.58	9.31	8.81	12.16	**3.22**
	20×20	28.27	21.81	22.89	17.29	31.61	17.27	15.17	16.74	15.59	14.69	22.61	10.87	9.95	9.04	11.54	**2.88**
	$30 \times 15^*$	35.04	24.07	22.89	21.14	33.00	18.52	17.09	23.23	18.96	17.30	24.35	13.98	10.97	**10.61**	14.13	10.66
	$30 \times 20^*$	33.45	24.82	26.75	22.64	33.62	21.47	18.49	23.59	19.94	20.44	28.40	16.09	13.43	**12.66**	16.35	13.18
	$50 \times 15^*$	24.01	16.45	17.59	15.89	22.38	12.23	10.06	14.14	13.51	13.52	17.06	9.32	5.47	**4.93**	11.01	12.25
	$50 \times 20^*$	25.59	17.84	16.79	20.34	26.51	13.24	11.55	17.65	14.57	13.96	20.44	9.89	8.35	**7.58**	11.25	13.59
	$100 \times 20^*$	14.05	8.33	8.71	13.49	13.61	5.86	5.85	10.39	7.02	7.11	13.30	3.96	2.32	**2.06**	5.90	10.97
	Avg	27.35	19.47	20.10	18.01	27.09	14.92	13.37	17.21	15.03	14.49	20.50	10.46	8.38	**7.78**	11.26	8.36
Demirkol's benchmark	20×15	27.94	27.54	30.68	24.18	39.03	-	17.99	17.17	22.28	23.85	29.27	19.43	12.00	11.32	-	**5.29**
	20×20	33.19	26.80	26.26	21.27	37.73	-	19.36	18.83	18.89	21.61	27.10	15.97	13.51	12.33	-	**4.69**
	$30 \times 15^*$	31.15	31.89	36.86	26.51	42.03	-	21.75	20.73	26.79	30.73	33.99	16.48	14.41	**14.01**	-	14.16
	$30 \times 20^*$	34.37	32.06	32.34	28.50	39.69	-	25.73	23.31	26.12	28.34	33.59	20.18	17.10	**15.80**	-	16.67
	$40 \times 15^*$	25.26	27.00	35.85	24.68	35.55	-	17.51	17.90	23.15	30.25	31.54	17.62	11.71	**10.91**	-	16.33
	$40 \times 20^*$	33.81	32.28	35.89	29.44	39.64	-	22.19	24.46	27.58	31.85	35.80	25.64	15.99	**14.81**	-	22.50
	$50 \times 15^*$	24.30	27.65	34.88	23.05	36.46	-	15.67	17.70	24.07	30.97	32.73	21.74	11.22	**10.64**	-	14.90
	$50 \times 20^*$	30.01	30.34	36.76	30.22	39.52	-	22.44	23.50	26.75	32.71	36.12	15.17	15.81	**15.00**	-	22.49
	Avg	30.00	29.45	33.69	25.98	38.71	-	20.33	20.44	24.46	28.79	32.52	19.03	13.97	**13.10**	-	14.63

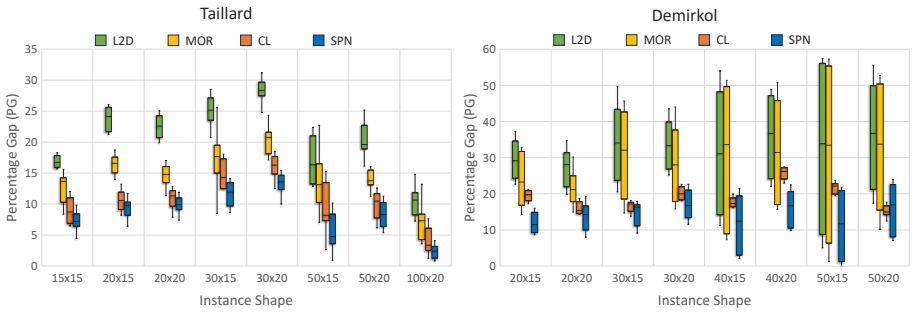

Figure 7: Percentage Gap statistics of the best randomized constructive approaches for various shapes of each benchmark and with $\beta = 128$.

19.50%[2]. Additionally, the SPN generalizes well to shapes larger than training ones (marked with *), as its gaps do not progressively increase on such shapes.

Notably, our SPN is competitive even when compared with non constructive approaches. Specifically, the SPN_{512} largely outperforms the enhanced metaheuristic NLS_A, and on shapes larger than 20×20, it always achieves lower gaps with just a few seconds of computations (see Sec. 4.4.2 for times) than a MILP executing for 3600 sec. We also remark that the SPN is better than the CP-based proposal hCP, which achieves an average makespan of 2670 and 5701 on Taillard and Demirkol benchmark respectively, while our greedy SPN yet obtains an average of 2642 and 5581. Therefore, we conclude that our proposed SPN and training strategy are effective in solving the JSP, despite their simplicity compared to models and training strategies adopted in the literature.

[2]Only qualitative results are disclosed about L2S [79], we did our best to report them.

Table 10: The average PG of greedy constructive approaches on the same two Taillard's instances of the ten available for each shape selected in [58, 80].

		Dispatching Rules				RL					
	Shape	SPT	MWR	MOR	INSA	TRL[58]	L2D[13]	CL[14]	DQN[80]	SPN	MILP
Taillard's benchmark	15×15	17.50	18.76	15.21	15.45	35.37	22.34	15.16	**7.11**	15.14	0.00
	20×15	29.69	24.52	26.83	15.25	32.10	26.83	17.85	**13.87**	13.99	2.35
	20×20	27.76	22.05	23.10	16.70	28.29	30.27	**16.16**	17.88	17.37	3.41
	$30 \times 15^*$	39.45	22.79	23.92	21.13	36.41	31.47	20.29	**16.13**	17.06	12.29
	$30 \times 20^*$	30.03	27.70	27.51	**19.17**	34.66	38.26	22.15	21.78	21.13	15.55
	$50 \times 15^*$	30.14	23.78	26.16	12.42	31.85	24.58	15.61	17.69	**11.82**	13.23
	$50 \times 20^*$	24.26	18.37	18.62	22.96	28.03	28.62	**14.19**	19.50	14.42	13.70
	$100 \times 20^*$	14.23	8.21	8.54	15.19	17.99	12.47	5.50	9.52	**4.96**	13.25
	Avg	26.63	20.77	21.24	17.28	30.59	26.85	15.86	15.44	**14.49**	9.22

To further demonstrate the quality of our SPN, we depict in Figure 7 the PG statistics of four randomized constructive approaches on the ten instances of each shape of the benchmarks. The box plot of Taillard's instances reveals the statistical dominance of our SPN as it shows lower median and quartiles compared to the other approaches. In Demirkol's plot, we observe in general larger variations for all the approaches but CL. However, despite experiencing larger variations than in Taillard's case, our SPN produces quality solutions resulting in low median values. It is worth noting that the SPN, the other algorithms, and also the MILP tend to struggle more with Demirkol's instances, compare e.g., the average PGs and variations of the 50×20 shape in both benchmarks. This is caused by the second half of the benchmark (from instance dmu40 to dmu80), where all algorithms but CL find worse solutions compared to the first half. As the first and second half comprise the same instance shapes, this evidence supports the observation on the complexity of Section 2.2.2, i.e., the complexity of JSP instances is not related only to their size but also to the distribution of operation processing times.

Finally, to comprehensively compare our SPN with all the reviewed ML proposals, we include in Table 10 the average PGs on the same two instances of each shape selected by TRL [58] and DQN [80] in Taillard's benchmark. On these subset of instances, we see that TRL is the worst-performing ML approach and that DQN roughly aligns with CL. As our SPN is always the best or second best algorithm on each shape, it achieves the lowest overall average gap (Avg) excluding the MILP, remarking once again the quality of our proposal.

4.4.2 Execution Times

We also assess the timing factor, an important aspect in some scheduling scenarios. To this end, we compare in Figure 8 the execution time of dispatching rules (PDR), the INSertion Algorithm (INSA), the SPN, and L2D [13] on typical shapes of the two benchmarks, when applied in a greedy constructive manner. We also include the execution time of our SPN when sampling 512 solutions (SPN_{512}) to demonstrate that sampling multiple solutions does not dramatically increase times. Note that all these algorithms were executed on the same

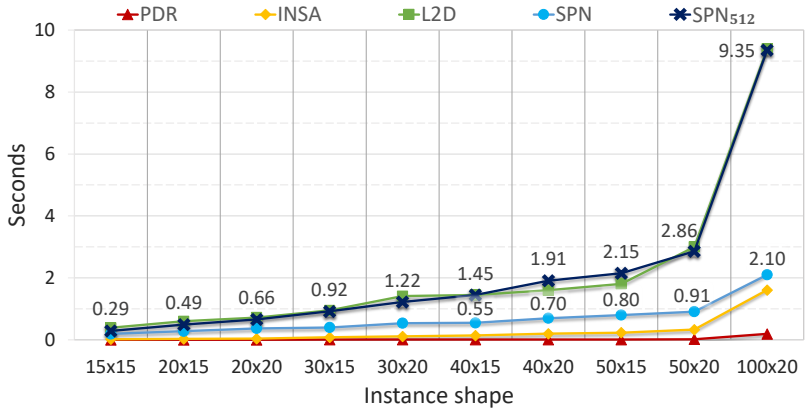

Figure 8: The execution times of algorithms on different instance shapes considered in Table 9. PDR, INSA, SPN, and L2D construct a single greedy solution, while SPN_{512} samples 512 parallel solutions.

machine described at the beginning of Section 4.4. We omit other RL proposals as they do not disclose execution times or is unclear whether they used GPUs.

When sampling 512 solutions, the SPN takes less than a second on medium shapes, less than 3 sec on large ones, and around 10 sec on the big 100×20. As these trends align with L2D, which only construct a single solution, our SPN is a much faster alternative, despite being also better in terms of quality. This is also confirmed by the execution time of the SPN when constructing a single greedy solution, which are slightly higher than INSA. Finally, we remark that the SPN and any ML-based approach are likely to be slower than PDRs and constructive heuristics, either applied in a greedy or randomized way. One can see that such approaches work faster: PDRs take 0.2 sec and INSA 1.61 sec on 100×20 instances. However, this increase in execution time is not dramatic and is largely justified by the better performance, especially in the case of our SPN.

4.4.3 The effect of β on Training

As we are proposing a new training strategy based on sampling, we assess the impact of sampling a different number of solutions β while training. To this end, we retrain a new SPN as described in Section 4.3 with a number of solutions $\beta \in \{32, 64, 128, 256\}$, where we stop at 256 as the memory usage with larger values becomes impractical. Then, we test the resulting models by sampling 512 solutions on all the instances of both benchmarks for a broader and unbiased assessment. Figure 9 reports the average PG (the lower the better) of the trained SPNs (different colored markers) on each benchmark shape. Finally, to ease the comparison with competitors, we also include the results of the best RL proposal in Table 9, namely CL (dashed line), and the MILP (dotted line).

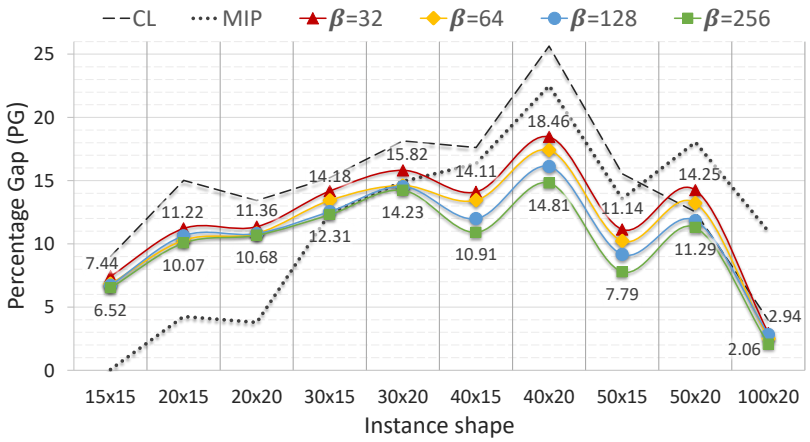

Figure 9: The performance of the SPN when trained by sampling β solutions. For each shape, we report the average PG on instances of both benchmarks by sampling 512 solutions during testing.

Overall, we see that training by sampling more solutions slightly improves the model's overall performance, as outlined by lower PGs for increasing β. We also observe that such improvement is less marked on shapes seen in training, such as in $15 \times 15, 20 \times 15$, and 20×20 shapes, while is more marked on others. This suggests that training by sampling more solutions results in better generalization, although it is more memory-demanding. However, the trends observed for the SPN in Table 9 remain consistent with smaller β, demonstrating the robustness of our self-labeling training strategy.

4.4.4 The effect of β on Testing

Finally, we assess how the number of sampled solutions β impacts the quality of the solution produced by our SPN at test time. To evaluate such an impact, we plot in Figure 10 the average PG on different shapes for varying $\beta \in \{32, 64, 128, 256, 512\}$. For this analysis, we use the SPN trained by sampling 256 solutions, the same used in Table 9. As done in Section 4.4.3, we also report the results of CL and MILP to ease the comparison.

Despite the reduced number of solutions, the SPN remains a better alternative than CL, one of the best RL proposals, and still outperforms the MILP on medium and large instances. Not surprisingly, by sampling more solutions the performance of the SPN improves, but this also increases the model execution times. Although we verified that sampling more than 512 solutions further improves results, we decided to stop at $\beta = 512$ as a good trade-off between performance and execution time.

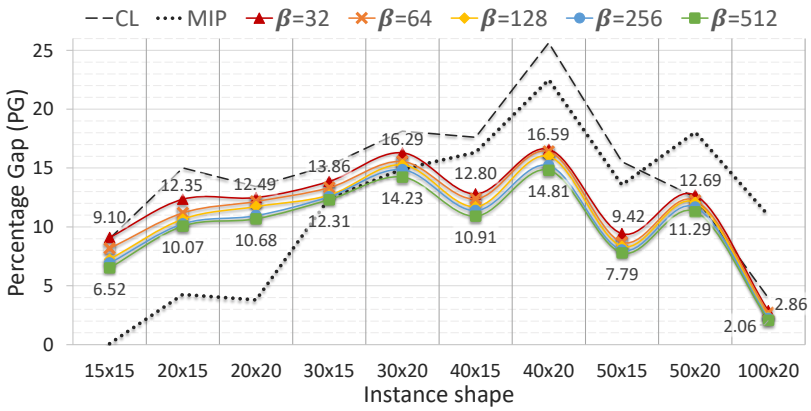

Figure 10: The performance of the SPN for varying numbers of sampled solutions β at test time. For each shape, we report the average PG on instances of both benchmarks.

4.5 Final Remarks

This work unveils a novel Self-Labeling strategy, which may effectively simplify the training of generative models, typically confined to more intricate reinforcement learning paradigms. Our innovative strategy, leveraging a classic Pointer Network and relying solely on model-generated solutions, not only outperforms established methods like PDRs and RL algorithms but also showcases remarkable resilience to varying parameters.

Looking ahead, our Self-Labeling strategy may be easily extended beyond the JSP, and we plan to apply it to diverse combinatorial problems such as routing ones. Furthermore, our training strategy can also serve as a pretext task for enhancing existing learning-based approaches, particularly those rooted in reinforcement learning. This work lays a solid foundation for future advancements in generative models, offering new avenues for solving complex scheduling and routing problems across various domains.

5 Solving a Complex Shop Problem

The focus of this section is on a shop scheduling problem arising from a real-world off-road vehicle manufacturing process. Off-road vehicles play a central role in modern and automated industries operating in sectors such as agriculture, construction, and logistics. The production of these vehicles often involves specialized components and systems that require expertise, dedicated tools, and long assembly and installation times, making the manufacturing process more intricate than in traditional vehicles. Additionally, customization and configuration based on specific customer needs are common and further increase the complexity of the production. Therefore, the development of effective scheduling methods that account for the characteristics of different vehicle types is crucial and can result in significant cost savings and increased productivity.

Herein, we focus on the scheduling problem encountered by a company that manufactures vehicles according to a *three-level assembly* process. At the first level, the vehicle chassis is equipped with all the necessary components (*e.g.*, wheels and engine), which are assembled in the two upper levels. The assembly process takes place in specific areas of the shop floor, known as *workcenters*, each of which houses one or more machines. The chassis moves through the workcenters of the first level in a fixed sequence that is identical for all vehicles. To perform any assembly operation, both a machine and a worker are required, where the number of workers is typically smaller than the number of machines, making them a *scarce resource*. The goal is to assign machines and workers to assembly operations in a way to minimize the total completion time of vehicles.

This manufacturing process has a unique and distinctive precedence structure among assembly operations forming a *directed rooted in-tree*, a special kind of directed tree characterized by a root vertex where all paths end. Therefore, we refer to this problem as the rooted in-Tree Resource Constrained Flexible Flow Shop Scheduling (rTRCFFSP).

To the best of our knowledge, no prior publications have addressed the rTRCFFSP. Nevertheless, this problem contains characteristics of several others. For instance, a rTRCFFSP sub-problem stemming from the identical sequence of workcenters at the first level is the *Flexible Flow Shop Problem*, which generalizes the classic Flow Shop, where operations at each stage can be processed

on one of several identical parallel machines (see [103, 104] for recent surveys). Additionally, the third and uppermost assembly level of the rTRCFFSP is a *Parallel Machine Scheduling Problem* with identical machines (see [105]). Finally, the constraint imposed by the scarce number of workers can be viewed as a *renewable resource* in a Project Scheduling Problem, (see, *e.g.*, [106]). These and other characteristics are further explained in Section 5.1.

To address the rTRCFFSP, we developed heuristic algorithms inspired by related scheduling problems, as solving the problem with standard MILP formulation is intractable even for small instances. Specifically, we evaluate priority dispatching rules when employed in an iterative and randomized manner, and we adapt job sequencing approaches commonly used in Flow Shop variants. Additionally, we propose a novel constructive algorithm building upon dispatching rules that is specifically designed to exploit the problem-specific structure of rTRCFFSP. The choice and the design of such algorithms are guided by a dual objective: on one side, we propose simple and effective algorithms appealing from an industrial perspective; on the other side, we seek to gain insights into the interplay of the unique combination of rTRCFFSP's characteristics.

Our proposals have been extensively evaluated on two benchmark sets mimicking industrial use cases, four configurations of the shop floor machines, and eight real-life scenarios. As the MILP formulation of the problem fails to produce quality lower bounds, we additionally propose a simple procedure for rapidly computing high-quality lower bounds for our benchmark instances. These lower bounds are thus used to evaluate the proposed algorithms. Our results demonstrate the effectiveness of both the job sequencing and the proposed constructive heuristic, with the latter exhibiting the best overall performance.

Note:

An extracted version of the work under the second round of revision at: Andrea Corsini, Mauro Dell'Amico, and Dario Bezzi. "*Lower and Upper Bounds for Scheduling a real-life Assembly Problem with Precedences and Resource Constraints*", in Computers & Operations Research, 2024.

5.1 The manufacturing problem

The manufacturing process involves the assembly of several components on a basic vehicle chassis, including, e.g., the engine, motion distribution group, wheels, and electronic components. Before installation on the chassis, components are prepared through assembly operations of sub-components, and sometimes, subcomponents may also require additional operations. Thus, the manufacturing process is composed of *three levels* of operations: two dedicated to component preparation and one to installation.

The process takes place on a large shop floor that is divided into *workcenters*, with each workcenter containing several identical parallel *machines*. Each assembly operation is associated with a specific workcenter and requires one of the machines of the workcenter to be executed. It is important to note that in

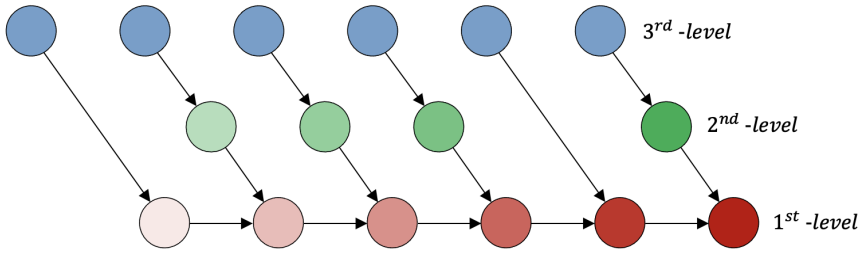

Figure 11: The precedences in a job. 3^{rd}-level operations (blue circles) are executed on the same workcenter; 2^{nd} and 1^{st}-level operations (respectively green and red circles) are executed on distinct workcenters.

reality, a machine may refer to a specific tool, an equipped workbench, or even a section of the floor where the chassis is placed before an operation is executed.

A crew of *human workers* with identical skills is responsible for carrying out the assembly operations. Any assembly operation requires exactly one machine in its workcenter and one worker. The set of human workers is finite and its cardinality is smaller than the number of machines on the shop floor, hence not all machines can be active at the same time. Additionally, workers can freely move among workcenters and from one machine to another.

The objective of this problem is to schedule the assembly operations required to produce a set of vehicles (frequently of different types) in the shortest amount of time possible, i.e., to minimize the *makespan*.

The assembly process of each type of vehicle maps on a *job*, and it consists of three levels of operations with the following structure:

1. **First-level operations**: refer to the installation of a component on the chassis and must be executed following an order identical for every vehicle. These operations always require as a prerequisite a single component, which is obtained either through a 2^{nd}-level or a 3^{rd}-level operation. Each 1^{st}-level operation is assigned to a distinct workcenter.

2. **Second-level operations**: produce the components that are installed on the chassis at the 1^{st}-level. These operations are optional and always require a single 3^{rd}-level component as a prerequisite. Each 2^{nd}-level operation is assigned to a different workcenter.

3. **Third-level operations**: transform raw material into a component that is used either by a 2^{nd}-level or a 1^{st}-level operation. These operations never require other operations to be performed before. All 3^{rd}-level operations are assigned to a unique workcenter.

This hierarchical structure among job operations is well represented with a directed rooted in-tree, where vertices are operations and arcs are precedences. In this tree, there is a vertex u, called *root*, and for any other vertex v there is

exactly one direct path from v to u. The 1^{st}-level operations form a backbone path to the root. In each vertex of the backbone enters an additional path with at most two vertices, starting from a leaf vertex corresponding to a 3^{rd}-level operation and possibly containing a vertex corresponding to a 2^{nd}-level operation. An example of this graph is depicted in Figure 11.

Real-life setting.

The real-life problem that inspired this work involves the scheduling of production orders with a varying number of jobs ranging from 20 to 30. The industry receives orders requiring to assemble vehicles (*jobs*) of different types. These orders are grouped together to form production batches (*instances*). Within each job, assembly operations can take up to several hours, thus the entire production of an order requires weeks to complete. Although the approximate time for executing an operation depends on the specific job, and hence of the type of vehicle, it goes in general from 30 minutes up to 13 hours. The variability caused by human factors and machines can be assumed negligible. The shop floor is divided into 11 workcenters for a total of 36 machines. At the 1^{st}-level, there are 6 workcenters with a number of machines given by the sequence $(4, 3, 2, 2, 3, 2)$, starting from the beginning to the end of the main path. At the 2^{nd}-level, there are 4 workcenters with 2 machines each, except for the last one, which has 4 machines. At the 3^{rd}-level there is a single workcenter with 10 machines. Lastly, a crew of 20 human workers is available to execute operations, limiting the maximum number of machines working in parallel to 20.

5.2 Related problems

We have already anticipated that the scheduling problem given by the 1^{st}-level operations is a *Flexible Flow Shop Problem* (FFSP), see [107, 108, 109]. The FFSP combines two scheduling problems: parallel machine scheduling and flow shop scheduling. At each stage of the flow shop, the FFSP presents a set of parallel machines, instead of a single one. The key decision of the parallel machine problem is the allocation of operations to machines whereas the key decision of the flow shop problem is the sequence of jobs through the shop. In the literature, this problem is also referred to as the *Hybrid Flow Shop Problem* (see, e.g., [104, 110]). In rTRCFFSP, we deal with a more complex structure since each operation of the flow shop has one or two additional predecessors to be scheduled (2^{nd}-level and 3^{rd}-level operations).

rTRCFFSP includes also a global constraint on human resources. A flow shop with unit processing times and a renewable resource has been studied in [111]. A generalization of the FFSP with limited human resources and setup times has been recently considered in [112] and solved with a modified Backtracking Search Algorithm hybridized with a Tabu Search. However, human resources are only dedicated to the activities related to the setup times and not to the execution of the production operations. Another version of the FFSP, related to our setting, is tackled in [113]. This paper studies in detail the effects

of human characteristics like skills, age, and learning/forgetting effects. The objective function is a weighted sum of the makespan and total flow time. A Particle Swarm Optimization algorithm is proposed to solve the problem.

The rTRCFFSP also relates to the *Resource-Constrained Project Scheduling Problem* (RCPSP) [114, 106]. The RCPSP asks to schedule project activities, described by a directed graph, which may require one or more resources for their execution. There is a large number of problem variants in the RCPSP literature. Some distinctive characteristics are: i) a resource can be *renewable*, as in rTRCFFSP, or *non-renewable* (i.e., once consumed a resource is no longer available for later activities); ii) one activity may require multiple resources, each in a certain amount; iii) resources can handle multiple activities at the same time. Each rTRCFFSP job defines a project in an RCPSP with identical renewable resources (workers and machines), each of which can handle one operation at a time. Therefore, an instance of the rTRCFFSP induces a *Multi-Project* problem, denoted as RCMPSP, see [106]. We address the reader to [115] for an overview of the principal variants and solution methods of the RCMPSP.

The rTRCFFSP also generalizes the scheduling of tasks with caterpillar precedence graphs on dedicated machines, namely the PD|caterpillar|$C_{\max}$. In [116] the authors present lower bounds and heuristic algorithms for this problem.

Scheduling problems where an operation can be processed only if a machine and a worker are simultaneously available are known as *Dual-Resource Constrained*. [117] and [118] present surveys on several types of shop scheduling problems with dual-resource constraints. However, the specific structure of the precedence graph and machine setting of rTRCFFSP do not match any of the problems listed in these two surveys.

5.3 Problem formulation

An instance of the rTRCFFSP asks to schedule a set of jobs $J = \{1, \ldots, n\}$ onto a set of workcenters $W = \{1, \ldots, m\}$. The set W is partitioned into three disjoint subsets: $W = W^1 \cup W^2 \cup W^3$, with $|W^1| \geq |W^2|$ and $|W^3| = 1$. Each workcenter $w \in W$ contains m_w identical parallel machines that can handle one operation at a time. Each job $j \in J$ is composed of a set of operations $O_j = O_j^1 \cup O_j^2 \cup O_j^3$. Operation $o \in O_j$ must be executed on one of the machines of workcenter $w(o) \in W$ for an uninterrupted amount of time $p(o) \in \mathbb{Z}_{\geq 0}$. (The choice of the machine to be used for each operation is part of the decision process.) The set of operations is organized as follows:

- O_j^1 is the sequence of 1^{st}-level operations, where an operation $o \in O_j^1$ must be executed on a machine of workcenter $w(o) \in W^1$;
- O_j^2 is the set of 2^{nd}-level operations, where $|O_j^2| \leq |O_j^1|$ and operation $o \in O_j^2$ must be executed on a machine of workcenter $w(o) \in W^2$;
- O_j^3 is the set of 3^{rd}-level operations that must be executed on a machine of the unique workcenter in W^3.

The operations in O_j must be executed by respecting the precedence relations of job $j \in J$ which are expressed through a directed rooted in-tree (see Section 5.1). Let $G_j = (V_j, A_j)$ be the digraph representing job j, where V_j contains one vertex for each operation in O_j and A_j is the set of precedence relations among the operations. We use $u_j \in V_j$ to identify the root of G_j. The arcs of A_j have no cost while each vertex $v \in V_j$ is given a cost $p_v = p(o)$, where $o \in O_j$ is the operation associated to vertex v. Moreover, let $w_v = w(o)$ denote the workcenter that must execute the operation o associated to v. For each vertex $v \in V_j$, we use $tail(v)$ to refer to the length of the unique path starting at v and ending in u_j (computed as the sum of the costs of the vertices in the path but p_v). Similarly, let $head(v)$ denote the length of the longest path ending in v again excluding p_v. The complete set of precedences of a rTRCFFSP instance is represented by the (disconnected) digraph $G = (V, A)$, where $V = \bigcup_{j \in J} V_j$ and $A = \bigcup_{j \in J} A_j$.

A set of $R^{\max} < \sum_{w \in W} m_w$ workers with identical skills is available to execute operations. In order to start the execution of an operation both a machine and a worker must be available, and the preceding operations, if any, must be completed. Preemption is not allowed, i.e., an operation o starting at time t must be uninterruptedly executed in the time interval $[t, t + p(o)]$.

Summarizing, a rTRCFFSP instance is completely described by the digraph G, the set of workcenters W, the number of machines in each workcenter, and the number of workers $R^{\max}$. rTRCFFSP asks to assign the operation associated to each vertex $v \in V$ to a machine of its workcenter w_v and to a time interval $[t_v, t_v + p_v]$ by guaranteeing that no more than $R^{\max}$ operations are simultaneously executed. The objective is to minimize the maximum completion time among all the operations: $\max_{v \in V} t_v + p_v$.

5.3.1 Mathematical model

We propose a mathematical formulation of the rTRCFFSP based on the time-indexed model in [119]. In order to simplify the writing, we introduce the set $\mathcal{V}_w = \{v \in V : w_v = w\}$, for each $w \in W$, which contains all the vertices whose operations are assigned to workcenter w. In addition, let $0, 1, \ldots, T$ be the discretized planning horizon. For each $v \in V$, we define the time interval where the operation associated to v can start as: $T_v = \{head(v), \ldots, T - tail(v) - p_v\}$. Our time-indexed model uses a continuous variable z to define the makespan, and binary decision variables x, where:

$$x_{vt} = \begin{cases} 1 & \text{if the operation associated to } v \text{ starts at time } t \\ 0 & \text{otherwise} \end{cases} \qquad v \in V, t \in T_v$$

$$\min\ z \tag{18}$$

$$\text{s.t.} \sum_{t \in T_{u_j}} t x_{u_j\, t} + p_{u_j} \leq z \qquad \forall j \in J \tag{19}$$

$$\sum_{t \in T_v} x_{vt} = 1 \qquad \forall v \in V \tag{20}$$

$$\sum_{v \in \mathcal{V}_w} \sum_{h=t-p_v+1}^{t} x_{vh} \leq m_w \qquad \forall w \in W, t = 0, \ldots, T \tag{21}$$

$$\sum_{t \in T_a} t x_{at} + p_a \leq \sum_{t \in T_b} t x_{bt} \qquad \forall (a,\, b) \in A \tag{22}$$

$$\sum_{v \in V} \sum_{h=t-p_v+1}^{t} x_{vh} \leq R^{\max} \qquad \forall t = 0, \ldots, T \tag{23}$$

$$x_{vt} \in \{0, 1\} \qquad \forall v \in V, t \in T_v \tag{24}$$

$$z \geq 0 \tag{25}$$

We minimize the makespan (18)-(19). Constraints (20) impose that each operation is executed once, constraints (21) ensure that no more than m_w machines are used in parallel in each workcenter $w \in W$. Constraints (22) impose the precedence constraints for each pair $(a,\, b) \in A$. Finally, constraints (23) impose that no more than $R^{\max}$ operations are executed simultaneously.

MILP implementation

The model (18-25) contains a pseudo-polynomial number of binary variables, which amount to $O(|V|T)$. In addition, constraints (21) and (23) introduce multiple inequalities for each time instant t.

To reduce the number of binary variables and also limit the introduction of constraints (21) and (23), we rapidly compute a small planning horizon T with a deterministic PDR prioritizing operations with the earliest start time (refer to Section 5.4.1 for more details on PDRs). We selected this priority rule as it results to be the best-performing one in the analysis of sections 5.6.1 and 5.6.4. The solution produced by this PDR is also used to warm start the model by loading it as the initial incumbent before starting the Branch & Bound.

Lastly, we add to the model the inequalities (23) as lazy constraints (available in the most popular MILP solvers). Specifically, the solver starts by disregarding the inequalities declared lazy, and once a feasible integer solution is found, it invokes a user defined separation procedure. At each invocation, this procedure adds inequality (23) to the smallest time instant violating the $R^{\max}$ constraint. We avoid applying this strategy to constraints (21) as they are violated often when modeled as lazy constraints, lowering performance.

5.3.2 Lower bounds

To evaluate the performance of heuristic algorithms of the next Section 5.4, we introduce two lower bounds for the rTRCFFSP problem: one accounting for the

precedence structure and one for the human resource constraint $R^{\max}$.

The first lower bound considers one workcenter at a time by disregarding the constraint on the human resources. For each workcenter $w \in W$, we compute the minimum time at which w can start to execute operations as: $hmin_w = \min_{v \in \mathcal{V}_w} head(v)$. Similarly, we compute the minimum total time required to complete all the operations following any operation in w as: $tmin_w = \min_{v \in \mathcal{V}_w} tail(v)$. In addition, we remark that a lower bound on the time needed to execute the operations of w can be obtained by solving an identical parallel machine scheduling problem $(P||C_{\max})$, see, e.g., [120]. However, since $P||C_{\max}$ is NP-hard, we use the L_2 lower bound defined in [120] instead of the optimal solution value. With these three information, we compute the lower bound LB_w given by workcenter w as:

$$LB_w = hmin_w + L_2(w) + tmin_w. \tag{26}$$

As starting operations of w before $hmin_w$, completing all the operations of the workcenter in less than $L_2(w)$, and completing all the operations following w in less than $tmin_w$ is impossible; it is proved that LB_w is a lower bound for the rTRCFFSP. Thus, an overall lower bound can be obtained with:

$$LB_{prec} = \max_{w \in W} LB_w. \tag{27}$$

The other lower bound focuses on the human resources that are needed to execute operations. In this bound, we relax the precedence constraint and assume that all the operations are assigned to a unique workcenter, say w_{Rmax}, with $R^{\max}$ parallel machines. Using again the bound L_2 from [120], we obtain:

$$LB_{res} = L_2(w_{Rmax}) \tag{28}$$

Similarly to LB_w, it is simple to prove the impossibility of executing all the operations in V before $L_2(w_{Rmax})$ by respecting all the rTRCFFSP constraints.

The final lower bound LB for the rTRCFFSP can thus be computed as:

$$LB = \max(LB_{prec}, LB_{res}) \tag{29}$$

5.4 Heuristics

This section describes the designed algorithms for tackling the rTRCFFSP. We start by presenting constructive algorithms based on popular dispatching rules, then, we show how we adapted job sequencing approaches to the rTRCFFSP problem. Finally, we introduce a novel heuristic building upon constructive algorithms that exploits the special structure of the problem.

5.4.1 Priority Dispatching Rules

Priority Dispatching Rule (PDR) are simple heuristics particularly suitable to deal with complex environments [110]. Motivated by their simplicity and speed,

PDRs have been extensively studied and employed in various scheduling problems, see, *e.g.*, [121, 110, 122]. For these reasons, and since PDRs are frequently used as baselines for comparing other algorithms, we propose and evaluate four PDR-based algorithms, each employing a different priority rule.

A PDR algorithm is a simple constructive heuristic that schedules one operation at a time selected according to a priority rule, often a simple rule of thumb checking characteristics of operations. At each iteration, our PDRs compute the minimum time instant t in which at least one operation can be scheduled by verifying the following two conditions: (a) there are less than $R^{\max}$ machine in use at time t; (b) there is at least one operation with all its predecessors completed before t and one idle machine that can process the operation. Once a time t is identified along with the operations that can start at t, the operation with the highest priority is scheduled with starting time equal to t. This process is repeated until all the operations are scheduled. It should be noted that in rTRCFFSP, an operation can only be scheduled when both a machine and a worker are available, which is different from classic scheduling problems.

As priority rules, we selected the following popular four:

i) *Earliest Start Time* (**EST**): prioritizes operations with the smallest "start time", computed as the maximum completion time of its predecessors.
ii) *Shortest Processing Time* (**SPT**): prioritizes operations with the least amount of processing time;
iii) *Most Operations Remaining* (**MOR**): prioritizes operations with the largest number of operations in the path from the vertex corresponding to the operation to the root of the graph;
iv) *Most Work Remaining* (**MWR**): prioritizes operations with the longest *tail* of the corresponding vertex.

The main drawback of PDRs is that their myopic reasoning may perform well in some instances and very badly in others. To obtain more robust algorithms, it is common to introduce randomization in the selection process. Therefore, we have modified the traditional PDR approach by randomly selecting the next operation to be scheduled from the first 5 operations with higher priority. Based on preliminary analysis, we found that selecting from the top five operations provides the best trade-off between exploration and exploitation when compared to selecting among the top $k \in \{3, 4, \ldots, 10\}$ operations. We run this randomized PDR several times on the same instance until the time limit has expired and we return the solution with the minimum makespan.

5.4.2 Job Sequencing

In contrast to PDRs, which schedule one operation at a time, job sequencing heuristics schedule a complete job at a time. These heuristics define a permutation of jobs according to some criteria, then, they iteratively select the next unscheduled job in the permutation and schedule all its operations. Examples of job sequencing heuristics can be found in [123] and [124]. As observed in [125], scheduling an entire job at once, instead of a single operation, can significantly

Algorithm 2 Job Sequencing - Iterated Greedy (Js-IG)

1: $\pi^* = \pi =$ Initial_Solution() ▷ Apply NEH.
2: **while** not "Stop Condition" **do**
3: $\quad \hat{\pi} =$ Deconstruct(π)
4: $\quad \pi' =$ Reconstruct($\hat{\pi}$)
5: $\quad$ **if** Accept($z(\pi')$) **then** $\pi = \pi'$
6: $\quad$ **if** $z(\pi') < z(\pi^*)$ **then** $\pi^* = \pi'$
7: **end while**

reduce the solution space that is explored. This reduction may result in faster algorithms, but also in a final solution of lower quality. Due to the effectiveness of job sequencing heuristics in Flow Shop problems [124, 110, 126], and as an important part of the rTRCFFSP is a FFSP, we evaluate two such heuristics.

In general, the design of job sequencing heuristics involves two degrees of freedom: (i) how to define the permutation of jobs; (ii) how to schedule operations of a job. Herein, we propose two algorithms of different complexity that differ in the way they generate job permutations. Instead of using deterministic rules to generate permutations, these algorithms rely on a randomized iterative approach similar to that of randomized PDRs. Both algorithms employ the same *deterministic PDR* to schedule the operations of a job, which works as described in the previous Section 5.4.1. After testing several priority rules for scheduling job operations, we found that MWR provided the best results for job sequencing heuristics. In the remainder, we describe how each algorithm generates permutations of jobs.

Our first algorithm, called **RAND**, simply tries several job permutations at random. It starts from a pure random permutation, and at each iteration, it shuffles the current permutation to create a new one. For each permutation, the job's operations are scheduled using the deterministic PDR algorithm. The algorithm executes until its time budget expires.

The second algorithm, named **Js-IG**, is based on the popular *Iterated Greedy* paradigm [126], which has proven effective for Flow Shop problems [124, 127]. Specifically, Js-IG builds upon the original proposal of [124] for the permutation Flow Shop, but without utilizing any local search strategy. The complete algorithm is outlined in Algorithm 2.

Js-IG begins by generating an initial permutation π of jobs using an adapted version of the classic NEH algorithm [128] for rTRCFFSP. The NEH heuristic starts with an empty schedule and arranges jobs in decreasing order of the total processing time. Following this order, each job is inserted in the current partial schedule by trying all the possible positions in the current permutation. The permutation giving the schedule with minimum makespan is selected and the next job is considered.

The final permutation π is the starting point of the iterative phase (steps 2-7), which executes the following three steps until the time budget expires.

step 3: procedure *Deconstruct* removes d jobs from the current permu-

tation π using a roulette-wheel selection, where the score of a job is its total idle time in the schedule: the higher the idle, the greater the probability of being removed.

step 4: procedure *Reconstruct* builds a new complete permutation π' by inserting each of the d removed jobs, following their removal order, in the position of the partial permutation that produces the minimum makespan.

step 5: procedure *Accept* receives the makespan $z(\pi')$ of the induced rTRCFFSP's solution π' and accepts it using the simulated annealing criterion of [124]. It always accepts an improving permutation, while it accepts a worsening one with probability $e^{(z(\pi)-z(\pi'))/T}$, with constant temperature:

$$T = \frac{\tau}{10 \sum_{j \in J} |O_j^1|} \sum_{v \in V} p_v, \tag{30}$$

where τ is a parameter that needs to be adjusted.

In all our experiments, we set $\tau = 0.4$ and $d = 3$. This pair of values was the one giving the best performance for $\tau \in \{0.2, 0.3, \ldots, 0.8\}$ and $d \in \{2, 3, \ldots, 6\}$ on a random sample of instances tested in Section 5.5.

5.4.3 A novel PDR Approach

Lastly, we propose a novel method, called **PDR-IG**, that generates priorities for a PDR algorithm by targeting the 1^{st}-level operations of the rTRCFFSP within an Iterated Greedy framework. As noted in Section 5.1, the sub-problem defined by 1^{st}-level operations is a Flexible Flow Shop Problem and is the more complex among the problems induced by the three levels. The rationale of this method is that the FFSP contains the operations in the backbone of the precedence graph, and this backbone induces the most important problem constraints.

PDR-IG uses solutions of this FFSP to define priorities for scheduling all the operations of the rTRCFFSP. More specifically, given a solution of the FFSP, let $C(u_j)$ be the completion time of the last operation of job j in this solution (see Section 5.3). We define the due date of each operation o of the complete problem as: $d_o = C(u_j) - tail(o)$. Then, the PDR prioritizes operations with the minimum value of the *Latest Start Time*: $l_o = d_o - p_o$. This PDR constructs a single solution according to the priorities as explained in Section 5.4.1 and without any randomization. In addition, to consider the existence of 2^{nd} and 3^{rd} levels operations, we extend the FFSP by setting as *release date* of each 1^{st}-level operation o its $head(o)$. Let rFFSP denote this extended problem.

To solve the rFFSP, we again adopted the framework by [124]. The complete PDR-IG is reported in Algorithm 3. In step 1, the algorithm considers the rFFSP consisting only of 1^{st}-level operations and solves it with a modified version of the NEH heuristic, which considers the release date constraint while scheduling operations. Step 2 uses procedure Complete to first define the priorities of the operations based on the current solution rFFSP, then runs the PDR algorithm with these priorities to obtain a solution of the complete problem.

Algorithm 3 PDR - Iterated Greedy (PDR-IG)

1: s_{rffsp} = Initial_Solution_rFFSP()
2: s^* = Complete(s_{rffsp})
3: **while** not "Stop Condition" **do**
4: $\quad \hat{s}_{\text{rffsp}}$ = Deconstruct(s_{rffsp})
5: $\quad s'_{\text{rffsp}}$ = Reconstruct($\hat{s}_{\text{rffsp}}$)
6: $\quad s'$ = Complete(s'_{rffsp})
7: $\quad$ **if** Accept(s') **then** $s_{\text{rffsp}} = s'_{\text{rffsp}}$
8: $\quad$ **if** $z(s') < z(s^*)$ **then** $s^* = s'$
9: **end while**

The while loop at steps 3-9 implements the Iterated Greedy method using procedures Deconstruct and Reconstruct to change the solution of rFFSP and again procedure Complete to obtain a rTRCFFSP solution. Deconstruct and Reconstruct are a modified version of the corresponding procedures from [110] obtained by taking into account the release date constraints. The Stop Condition and the Accept criterion are the same of the Js-IG.

5.5 Experimental setup

The proposed algorithms are evaluated on synthetic benchmark instances that emulate industrial use cases, and eight real-life scenarios. Initially, we attempted to solve all our instances using the MILP approach presented in Section 5.3.1 with Gurobi 9.5.1. For instances with up to 10 jobs, we were able to compute optimal solutions in less than a CPU hour; however, with 12 jobs the MILP required more than 2 hours, and with more jobs it slowly increases lower bounds by hardly improving over the warm start solution after 6 hours. As a result, the MILP approach failed to provide any useful information, such as optimal solutions or lower bounds.

In real environments, we need to schedule 20-30 jobs at a time, therefore, we conducted our experiments with only the heuristic algorithms. Due to the lack of optimal solutions, we evaluate our algorithms in terms of the Percentage Gap (PG) from the lower bound presented in Section 5.3.2. Formally, given a single instance, let C_{LB} be the lower bound on the makespan, and C_{max} the makespan of a heuristic. The PG is defined as:

$$\text{PG} = \left(\frac{C_{max} - C_{LB}}{C_{LB}} \right) * 100 \tag{31}$$

All the algorithms of Sections 5.4.1-5.4.3 have been written in C++, compiled with g++ 9.3.0, and executed on an Ubuntu machine equipped with an Intel Core i9-11900K. These heuristics iteratively execute until a time limit is reached by taking in input a rTRCFFSP instance and returning the solution with minimum makespan. To guarantee reproducible results and fairness, we execute all the algorithms with the same initial random seed.

Table 11: The processing time of operations in each job level for each type of instance. **S** stands for Small, **M** for Medium, and **L** for Large.

Type	1^{st}-level			2^{nd}-level			3^{rd}-level		
	S	**M**	**L**	**S**	**M**	**L**	**S**	**M**	**L**
0	×	·	·	×	·	·	×	·	·
1	×	·	·	×	·	·	·	×	·
2	×	·	·	×	·	·	·	·	×
3	×	·	·	·	×	·	×	·	·
4	×	·	·	·	×	·	·	×	·
5	×	·	·	·	×	·	·	·	×
6	×	·	·	·	·	×	×	·	·
7	×	·	·	·	·	×	·	×	·
8	×	·	·	·	·	×	·	·	×
9	·	×	·	×	·	·	×	·	·
10	·	×	·	×	·	·	·	×	·
11	·	×	·	×	·	·	·	·	×
12	·	×	·	·	×	·	×	·	·
13	·	×	·	·	×	·	·	×	·
14	·	×	·	·	×	·	·	·	×
15	·	×	·	·	·	×	×	·	·
16	·	×	·	·	·	×	·	×	·
17	·	×	·	·	·	×	·	·	×
18	·	·	×	×	·	·	×	·	·
19	·	·	×	×	·	·	·	×	·
20	·	·	×	×	·	·	·	·	×
21	·	·	×	·	×	·	×	·	·
22	·	·	×	·	×	·	·	×	·
23	·	·	×	·	×	·	·	·	×
24	·	·	×	·	·	×	×	·	·
25	·	·	×	·	·	×	·	×	·
26	·	·	×	·	·	×	·	·	×

5.5.1 Benchmarks

To comprehensively evaluate our algorithms, we have generated two benchmark sets and four configurations of the shop floor.[1]

Typed instances

The first benchmark set, called *Typed*-Set, consists of 24 types of instances, where each type is defined by the intervals used to sample the processing times of operations at each level. We loosely defined three intervals for the processing times: Small $= [5, 50]$, Medium $= [30, 100]$, and Large $= [80, 150]$. The intervals of all the 24 instance types are given in Table 11. For each type, we generated 10 instances by uniformly sampling the processing time of job operations at the 1^{st}-level, 2^{nd}-level, and 3^{rd}-level in the corresponding intervals. This resulted in a total of 270 instances, with each instance consisting of 30 jobs. When a level of a type is associated with more than one interval, their processing times are sampled from the union of the intervals. As an example, a *Type15* instance has all the 30 jobs with 3^{rd}-level processing times uniformly sampled in $[5, 50]$, 2^{nd}-level times sampled in $[80, 150]$, and 1^{st}-level in $[5, 100]$.

Mixed instances

The second benchmark set, called *Mixed*-Set, has been designed to introduce more diversity in the jobs of the instances. It contains 200 instances with 30 jobs each. The first 100 instances are generated by *uniformly* deciding a type for each job from Table 11, and generating the processing times accordingly. The next 100 instances contain *batches* of the same jobs. To generate these instances, we randomly select an instance type and an integer $b \in [1, 10]$, we generate a batch of b jobs according to the current type, and we repeat the type and batch size selection until we have generated all the 30 jobs (in the last batch we possibly generate less than b jobs to avoid exceeding the total of 30).

[1] Instances and lower bounds are available at: https://github.com/AndreaCorsini1/inTreeFFSP

Table 12: Different machine configurations of the shop floor.

Config.	3^{rd}-level	2^{nd}-level				1^{st}-level						Tot
	w_1^3	w_1^2	w_2^2	w_3^2	w_4^2	w_1^1	w_2^1	w_3^1	w_4^1	w_5^1	w_6^1	
1	10	2	2	2	4	4	3	2	2	3	2	36
2	16	2	2	2	2	2	2	2	2	2	2	36
3	6	3	3	3	3	3	3	3	3	3	3	36
4	8	4	4	4	4	2	2	2	2	2	2	36

Shop floor configurations

We generated four machine configurations to evaluate different shop floor characteristics. Table 12 describes the configurations by reporting, in each row, the number of parallel machines that we include in the 11 workcenters (see Section 5.3). Note that the first configuration corresponds to the machine's setting of the real-case that inspired this work.

5.6 Results

5.6.1 Typed Instances

We start the evaluation of the algorithms by considering the *Typed*-Set of instances and the first configuration for the shop floor. Figure 12 reports the average Percentage Gap (PG) over all the 270 instances for each algorithm by varying the running time from 1 to 150 seconds. Remember that the PG is computed as given in Eq. 31 and thus compares the makespan of a solution with the proposed lower bound (see Section 5.3.2) of an instance.

Figure 12 highlights a clear dominance of the job sequencing and the PDR-IG heuristics with respect to the dispatching rules. We observe improvements up to three orders of magnitude in the PG. For this reason, we decided to discard the dispatching rules in the next experiments (we only report their results on real-life scenarios in Section 5.6.4). We can also see that the best overall proposal is the PDR-IG algorithm, followed by the job sequencing algorithm Js-IG and the RAND heuristic.

To provide a more accurate evaluation of these three algorithms, we present their performance for each of the 27 instance types in Table 13. For each type and algorithm, we report the average (Avg) PG, the standard deviation (Std), and the number of best solutions (# bst). The results in the table confirm that RAND has the worst performance, as it never produces average gaps better than Js-IG and PDR-IG. In addition, we see that on certain types of instances, Js-IG obtains slightly better average results than PDR-IG. However, when Js-IG fails to find the best solutions, its results are significantly worse than those of PDR-IG. Hence, the overall result of PDR-IG on the entire benchmark constantly outperforms that of Js-IG.

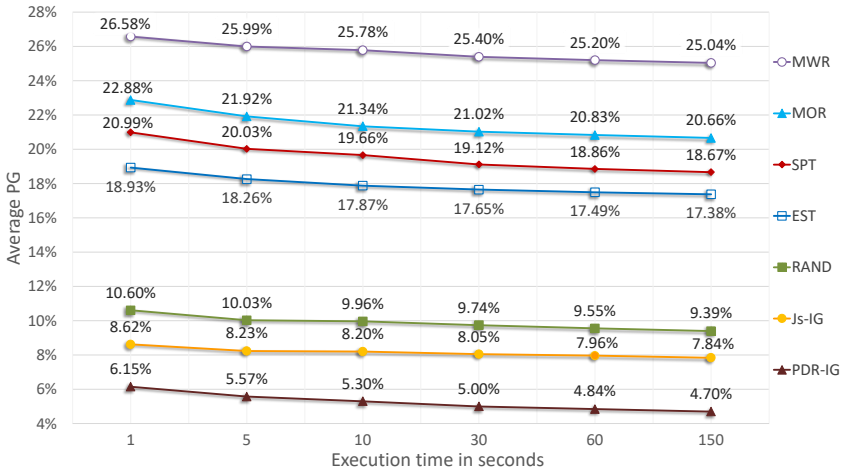

Figure 12: The average Percentage Gap of the algorithms on the 270 instances for time limits up to 150 seconds.

Table 13: Average Percentage Gaps from the lower bounds of typed instances (machine configuration 1) after 150 seconds of running time. The values in **bold** are the best across an instance type (row).

Type	Job Sequencing				Decomposition	
	RAND		Js-IG		PDR-IG	
	Avg (Std)	# bst	Avg (Std)	# bst	Avg (Std)	# bst
0	15.14 (0.77)	0	12.01 (1.11)	0	**7.53** (**0.72**)	**10**
1	5.01 (1.62)	0	**1.68** (1.48)	**8**	1.76 (**1.30**)	5
2	3.14 (1.32)	0	**1.26** (**0.77**)	**8**	1.54 (0.82)	2
3	1.56 (0.66)	0	**0.87** (**0.50**)	**7**	0.92 (0.51)	3
4	11.82 (3.24)	0	8.80 (3.45)	0	**5.86** (**3.02**)	**10**
5	2.44 (1.03)	0	**0.83** (**0.45**)	**7**	0.88 (0.62)	6
6	0.73 (0.49)	0	**0.46** (**0.38**)	**8**	0.63 (0.54)	6
7	1.87 (0.71)	0	**0.73** (**0.47**)	**7**	1.07 (0.68)	4
8	7.80 (1.24)	0	**5.04** (**1.17**)	**9**	5.97 (1.40)	1
9	7.86 (0.62)	0	7.36 (0.67)	0	**2.84** (**0.40**)	**10**
10	20.69 (0.97)	0	17.96 (1.00)	0	**10.53** (**0.93**)	**10**
11	3.02 (1.12)	0	**1.20** (**0.65**)	**9**	1.44 (0.77)	4
12	11.87 (**0.62**)	0	9.70 (0.91)	0	**6.41** (0.71)	**10**
13	15.85 (0.45)	0	13.60 (0.74)	0	**8.97** (**0.30**)	**10**
14	6.90 (1.49)	0	5.01 (1.57)	0	**1.68** (**0.90**)	**10**
15	1.06 (0.72)	0	**0.88** (**0.59**)	**7**	1.18 (0.76)	3
16	6.35 (2.93)	0	4.73 (2.84)	0	**3.12** (**2.07**)	**10**
17	18.02 (1.25)	0	16.23 (1.01)	0	**11.21** (**0.95**)	**10**
18	3.90 (0.66)	0	3.67 (0.63)	0	**1.62** (**0.27**)	**10**
19	10.98 (1.18)	0	10.49 (1.20)	0	**3.72** (**0.35**)	**10**
20	18.12 (1.59)	0	16.65 (**1.52**)	0	**8.18** (1.69)	**10**
21	5.35 (0.84)	0	5.09 (0.86)	0	**2.10** (**0.39**)	**10**
22	14.53 (1.16)	0	13.23 (1.18)	0	**4.57** (**0.50**)	**10**
23	20.63 (**0.39**)	0	19.05 (0.54)	0	**12.04** (0.40)	10
24	8.52 (0.40)	0	7.74 (**0.39**)	0	**2.40** (0.40)	**10**
25	13.35 (0.92)	0	11.85 (0.91)	0	**7.96** (**0.28**)	**10**
26	17.04 (0.55)	0	15.62 (0.65)	0	**10.58** (**0.28**)	**10**
Tot	253.56 (28.94)	0	211.75 (27.66)	70	**126.76** (**21.79**)	**214**

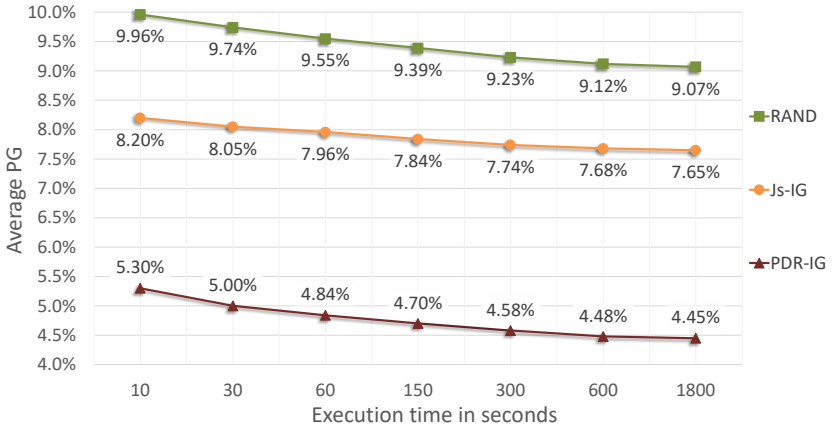

Figure 13: The average Percentage Gap on the 270 instances for each time limit.

5.6.2 Performance for Large Computing Times

To better assess the complete behavior of the three best algorithms (RAND, Js-IG, and PDR-IG); we run them again on the typed instances with increasing time limits. More specifically, we added to the previous computations the results obtained with a maximum running time equal to 300, 600, and 1800 seconds. Figure 13 resumes the results with time limits from 10 to 1800 seconds.

The three algorithms exhibit a clear ranking: RAND is the worse method with a PG of 9.07% at 1800 seconds, followed by Js-IG with a PG of 7.65% and by PDR-IG with 4.45%. Overall, we see that all the algorithms benefit from increasing running time, but the gains diminish after 150 seconds as doubling the time results in always lower improvements. Since the relative differences between proposals remain more or less consistent after 150 seconds, we choose to keep this time limit as a reasonable trade-off between performance and execution time for continuing the evaluation of the algorithms on the other instances and shop floor configurations.

5.6.3 Performance Under Different Shop Floor Configurations

To further extend our analysis on the *Typed* instances, we run the RAND, Js-IG, and PDR-IG algorithms on the other shop floor configurations not considered in previous sections, i.e., configurations 2, 3, and 4 of Table 12. In Table 14, we report the average PG and the number of best solutions obtained by each algorithm for each instance type and configuration, and in the last row, we accumulate the average PG and the number of best solutions across all types.

Similarly to configuration 1, RAND results to be the worst-performing algorithm, while the PDR-IG confirms to be the best algorithm, both in terms of the total average PG and number of best solutions.

Table 14: Average Percentage Gaps from the lower bounds of *Typed* instances and machine configurations 2-4, after 150 seconds of running time. The values in **bold** are the best across an instance type (row).

Type	Config. 2						Config. 3						Config. 4					
	RAND		Js-IG		PDR-IG		RAND		Js-IG		PDR-IG		RAND		Js-IG		PDR-IG	
	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst
0	13.44	0	10.97	0	**6.91**	**10**	10.73	0	10.42	0	**2.27**	**10**	20.27	0	16.12	0	**8.93**	**10**
1	7.56	0	**2.81**	**6**	3.17	4	26.65	0	22.55	0	**13.75**	**10**	7.88	0	**2.57**	**7**	2.82	4
2	4.98	0	**2.32**	**9**	2.86	2	10.47	0	8.37	0	**5.67**	**10**	4.94	0	**2.17**	**10**	2.58	0
3	1.64	0	0.89	**7**	**0.82**	5	15.26	0	12.02	0	**7.83**	**10**	11.82	0	9.54	0	**5.70**	**10**
4	11.71	0	8.01	1	**6.71**	**9**	17.79	0	14.61	0	**9.66**	**10**	12.18	0	8.79	0	**4.59**	**10**
5	5.05	0	**2.47**	**8**	2.88	2	20.33	0	18.06	0	**11.90**	**10**	4.63	0	**2.06**	**10**	2.54	0
6	0.66	0	**0.39**	**9**	0.54	8	5.28	0	**3.51**	**8**	4.04	2	8.51	0	6.83	0	**4.70**	**10**
7	2.11	0	**0.81**	**10**	1.14	0	14.15	0	12.16	0	**9.00**	**10**	15.33	0	13.38	0	**8.68**	**10**
8	8.86	0	**5.71**	**10**	6.51	0	18.72	0	16.93	0	**11.24**	**10**	6.21	0	3.91	2	**3.05**	**8**
9	8.95	0	7.59	0	**4.40**	**10**	3.32	0	3.19	0	**0.81**	**10**	5.50	0	5.20	0	**1.88**	**10**
10	16.06	0	14.13	0	**8.26**	**10**	10.58	0	10.39	0	**1.82**	**10**	16.82	0	15.52	0	**5.77**	**10**
11	5.15	0	**2.44**	**10**	2.95	2	23.55	0	22.82	0	**7.22**	**10**	5.85	0	2.89	5	**2.68**	**6**
12	7.85	0	6.59	0	**5.16**	**10**	5.00	0	4.89	0	**0.96**	**10**	7.55	0	7.24	0	**2.40**	**10**
13	14.16	0	12.58	0	**7.77**	**10**	11.41	0	11.21	0	**2.33**	**10**	20.34	0	17.62	0	**10.03**	**10**
14	6.56	0	4.43	1	**3.66**	**9**	27.32	0	24.71	0	**12.51**	**10**	8.93	0	6.87	0	**3.17**	**10**
15	1.16	0	**0.82**	**7**	0.91	4	7.37	0	7.23	0	**0.98**	**10**	12.04	0	10.23	0	**5.36**	**10**
16	5.38	0	3.90	4	**3.09**	**7**	14.24	0	13.28	0	**4.38**	**10**	14.71	0	12.68	0	**8.24**	**10**
17	16.59	0	15.18	0	**10.26**	**10**	21.41	0	19.14	0	**11.27**	**10**	18.85	0	16.84	0	**10.87**	**10**
18	7.14	0	6.17	0	**3.82**	**10**	1.75	0	1.66	0	**0.46**	**10**	2.78	0	2.61	0	**0.96**	**10**
19	12.90	0	11.63	0	**6.72**	**10**	5.18	0	5.05	0	**1.07**	**10**	7.96	0	7.66	0	**2.44**	**10**
20	12.01	0	11.09	0	**4.93**	**10**	12.27	0	12.06	0	**2.11**	**10**	18.07	0	17.51	0	**6.73**	**10**
21	6.64	0	5.85	1	**4.59**	**9**	2.43	0	2.36	0	**0.64**	**10**	3.79	0	3.62	0	**1.30**	**10**
22	12.08	0	10.97	0	**6.05**	**10**	6.26	0	6.10	0	**1.07**	**10**	9.49	0	9.15	0	**2.77**	**10**
23	17.34	0	16.24	0	**10.14**	**10**	12.03	0	11.82	0	**2.34**	**10**	18.55	0	17.49	0	**7.27**	**10**
24	7.11	0	**6.24**	**9**	6.55	3	3.77	0	3.69	0	**0.58**	**10**	5.61	0	5.43	0	**1.39**	**10**
25	10.56	0	9.52	1	**6.72**	**9**	6.91	0	6.79	0	**1.27**	**10**	10.45	0	10.01	0	**3.36**	**10**
26	15.44	0	14.30	0	**8.97**	**10**	12.92	0	12.71	0	**2.94**	**10**	21.03	0	19.27	0	**11.38**	**10**
Tot	239.09	0	194.06	93	**136.53**	**193**	327.09	0	297.73	8	**130.09**	**262**	300.07	0	253.18	34	**131.60**	**238**

Table 15: Results for *Mixed* instances and machine configurations 1-4, after 150 seconds of running time. The values in **bold** are the best across an instance type (row).

Type	Config. 1						Config. 2					
	RAND		Js-IG		PDR-IG		RAND		Js-IG		PDR-IG	
	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst
Uniform	11.3	0	6.4	41	**6.3**	**59**	9.5	0	5.5	29	**4.8**	**71**
Batched	12.9	0	8.5	31	**7.4**	**70**	11.3	0	7.7	24	**6.5**	**76**
Tot	24.1	0	14.9	72	**13.7**	**129**	20.8	0	13.2	53	**11.5**	**147**

Type	Config. 3						Config. 4					
	RAND		Js-IG		PDR-IG		RAND		Js-IG		PDR-IG	
	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst	Avg	#bst
Uniform	6.1	0	4.4	5	**2.6**	**92**	14.9	0	8.4	2	**5.8**	**97**
Batched	8.8	0	6.3	9	**3.4**	**94**	15.2	0	10.0	6	**7.9**	**95**
Tot	14.9	0	10.7	14	**6.0**	**186**	30.1	0	18.3	8	**12.6**	**192**

Table 16: Makespan of solutions for the eight real-life instances.

Name	Num. jobs	LB	Manual Solution	Dispatching Rules							MILP
				MWR	MOR	SPT	EST	RAND	Js-IG	PDR-IG	[LB, UB]
I1	17	761	952	1029	1026	940	899	924	924	**873**	[574, 947]
I2	21	905	1116	1245	1246	1114	1079	1093	1093	**1042**	[688, 1140]
I3	18	797	988	1066	1119	983	945	966	966	**913**	[685, 997]
I4	24	1013	1239	1406	1426	1253	1213	1216	1216	**1165**	[665, 1286]
I5	23	977	1200	1339	1375	1207	1163	1182	1182	**1130**	[681, 1254]
I6	25	1049	1268	1471	1460	1282	1246	1233	1234	**1195**	[833, 1326]
I7	22	941	1143	1285	1272	1149	1103	1102	1101	**1061**	[709, 1182]
I8	25	1049	1267	1493	1460	1294	1246	1230	1229	**1184**	[687, 1289]

We finally conclude our computational analysis on randomly generated instances by considering the *Mixed*-Set (see Section 5.5.1) and the four shop floor configurations. Table 15 reports the average PG and number of best solutions obtained by each algorithm on the 100 uniform instances and on the 100 batched instances, for each configuration. We remark again that PDR-IG achieves consistently lower average PG than Js-IG regardless of the shop floor configurations. Therefore, PDR-IG remains the best overall proposal also in mixed instances.

5.6.4 Real-life Instances

Due to internal company policies and regulations, we were allowed to use only eight real-life instances and their relative manual solutions for reporting. These instances involve the assembly of approximately 20-25 vehicles from five different jobs, and the processing times of these jobs are distributed similarly to type 17 of Table 11. The assembly of jobs is performed by a crew of 20 workers on the shop floor corresponding to the first machine configuration in Table 12. The manual solutions are constructed by scheduling one job at a time, in order of their arrival time, and occasionally job exchanges may be done by a local expert without any precise or reproducible logic.

Table 16 presents the makespan of the solution produced by the proposed algorithms on the instances I1,. . . , I8. The considered algorithms (columns) are the manual solution; the four dispatching rules; and the three heuristics: RAND, Js-IG, and PDR-IG. All these algorithms were executed as in previous sections for 150 seconds. We also consider the MILP formulation of the rTRCFFSP presented in Section 5.3.1, which was implemented in Gurobi 9.5.1 and executed for 3600 seconds.

It is noteworthy that the manual solution outperforms the MWR, MOR, and SPT rules, but is outperformed by the EST rule. Surprisingly, RAND and Js-IG algorithms perform worse than EST for I1-I5 instances, while they exhibit similar performance for I6-I8. The PDR-IG algorithm still consistently achieves the best solution across all instances, demonstrating consistent performance also in these real scenarios. Lastly, we underline that the presented MILP formulation

is practically useless even in these small instances, as both its produced bounds are dominated by our proposed lower bound and algorithms.

5.7 Final Remarks

This work introduces a new complex scheduling problem encompassing characteristics of other well-known problems. We not only formalized the problem, but also proposed baseline heuristics, generated a comprehensive set of benchmark instances, and introduced a lower bound for evaluating our algorithms and potentially future ones.

Although several methodologies exist to tackle related problems, our investigation revealed that many of them face challenges when applied directly to address the rTRCFFSP complexities. Therefore, we believe that this new problem may offer a valuable opportunity to extend concepts and modify assumptions inherent in existing algorithms for shop scheduling problems. Lastly, the multiple intertwined constraints of the rTRCFFSP pose serious obstacles to the adoption of standard exact methods, another opportunity for further research.

6 Parallel Drone Scheduling Vehicle Routing Problems with Collective Drones

The employment of drones in last-mile delivery is considered extremely strategic as leading distribution operators are facing a continuously increasing volume of parcels to handle, mainly generated by e-commerce (Statista, [129]). Considering that drones use low-emission electric motors, do not move along the road network, can fly approximately in straight lines, and are not affected by traffic congestions; their adoption for deliveries could lead to advantages for companies (operational costs reduction), customers (faster deliveries), and the whole society (sustainability). Forbes [130] refers to the heavy interest in drone technology as the "Drone Explosion". The authors of [131] forecast that autonomous vehicles will deliver about 80% of all parcels in the upcoming decade. Therefore, we analyze a mixed routing and scheduling problem where drones are used in conjunction with trucks for last-mile delivery.

Murray and Chu [21] introduced the idea of a new problem in which a truck and a drone collaborate to make deliveries. The authors present two new prototypical models expanding from the traditional Traveling Salesman Problem (TSP) called the Flying Sidekick TSP (FSTSP) and the Parallel Drone Scheduling TSP (PDSTSP). In both cases, a truck and some drones collaborate to deliver parcels. In the former model, drones can be launched from the truck during its tour, while in the latter one, drones are only operated from the central depot, and the truck executes a traditional delivery tour. In the remainder, we will focus on the latter problem, addressing the interested reader, e.g., to [132] and [133] for details and solution strategies for the FSTSP.

More formally, in the PDSTSP there is a truck that can leave the depot, serve a set of customers, and go back to the depot. In parallel, there is also a set of drones, and each one of them can leave the depot, serve a customer, and return to the depot before serving other customers. Some of the customers cannot be served by the drones, either due to their location or the characteristics of their parcel. The objective of the optimization is to minimize the makespan, i.e., the time of the last vehicle returning to the depot while serving all its customers.

A first Mixed Integer Linear Programming model for the PDSTSP was proposed in [21] together with some simple heuristic methods. Another MILP model and the first meta-heuristic, based on a two-step strategy embedding a dynamic programming component, were discussed in [134]. Another two-step approach was presented in [135], a hybrid ant colony optimization meta-heuristic was discussed in [136], and a variable neighbor search in [137]. In [138], an effective constraint programming approach was proposed, which optimally solved all the benchmark instances previously adopted in the literature for both exact and heuristic methods. Recently, in [139] another exact approach based on branch-and-cut was proposed, together with new benchmark instances.

Several PDSTSP variants were also introduced and studied in the literature, see e.g., [140] and [141] for extensive surveys. We review herein only those extensions of the PDSTSP that we find more relevant to the present study.

Recently, [142] discussed the *Parallel Drone Scheduling Multiple Traveling Salesman Problem*, which is a straightforward extension of the PDSTSP where multiple trucks are employed and the target is to minimize the time required to complete all the deliveries. The authors proposed a hybrid meta-heuristic, a MILP model, and a branch-and-cut approach. The same problem was independently introduced in [143], where the authors proposed three MILP models, together with a branch-and-price approach. A heuristic version of the branch-and-cut method was also introduced, aiming at solving the larger instances. A more realistic version of the PDSTSP was introduced in [144], where concepts such as capacity, load balancing, and decoupling of costs and times are taken into account. The authors proposed a MILP model and a *Ruin & Recreate* meta-heuristic. Constraint Programming methods for PDSTSP variants employing several trucks were also discussed in [145], with convincing experimental results.

One common assumption in the literature on combined truck-drone delivery models has been a linear battery consumption for drones, leading to fixed operation ranges and carrying capacities. Recently, power consumption models with more realistic settings have been presented, e.g., in [143], [146], and [147]. A review of drone energy consumption models is also available in [148]. In the patent [149] filed by Amazon Inc., a novel method using a so-called "Collective Drone" (c-drone) is introduced. Under this setting, multiple drones can be coupled to aerially transport items of large size and weight. By sharing resources, such as power and operating instructions, a collective drone might outperform a single drone. The authors of [22] joined the new concept of c-drone with advanced power consumption models to create an innovative problem, called the PDSTSP-c, where c stands for *collective*. In this problem, a realistic model is used to calculate the endurance, speed, and capacity of groups of drones working together to carry out tasks. Based on these calculations, they proposed a MILP model and a *Ruin & Recreate* meta-heuristic for this new problem.

An example of a PDSTSP-c instance is provided in Figure 14.

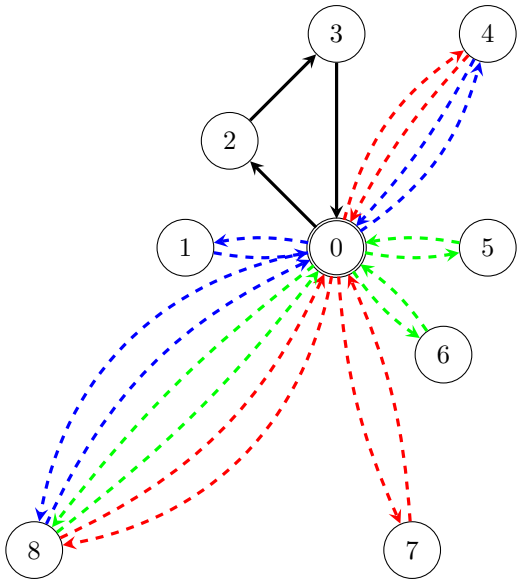

Figure 14: Example of a PDSTSP-c instance. Node 0 is the depot, the other nodes are customers. Travel times are omitted for the sake of simplicity. The black continuous arcs represent the tour of the truck $(0, 2, 3, 0)$. The dashed arcs depict the missions of the drones, each colour representing a different one. Notice that some of the missions are carried out by multiple drones.

Note:

An extracted version of the work published at:
Roberto Montemanni, Mauro Dell'Amico, and Andrea Corsini. "*Parallel Drone Scheduling Vehicle Routing Problems with Collective Drones*", in Computers & Operations Research, Volume 163, 2023. DOI: 10.1016/j.cor.2023.106514.

6.1 Parallel Drone Scheduling Traveling Salesman Problem with Collective Drones

In this section, we formally describe the PDSTSP-c, as originally introduced in [22], and present a new Constraint Programming model. We start from the single-vehicle problem as its description helps in introducing the multiple-vehicle version.

6.1.1 Problem Description

Let $G(V, E)$ be the complete graph describing an instance of the PDSTSP-c, with $V = \{0, 1, \ldots, n\}$ being the set of vertices where vertex 0 represents the depot and the remaining all the customers to visit (set $C = V \setminus \{0\}$). Each customer i requests the delivery of a parcel of weight w_i from the depot. The fleet of delivery vehicles is composed of a single driver-operated truck, with unlimited range and capacity, and a set D of m homogeneous drones that are based at the depot and equipped with batteries of given capacity (a fresh battery is installed before each mission).

The truck performs its task within a single tour, beginning from the depot, visiting all its assigned customers, and returning to the depot. The truck travel times between pairs of vertices $i, j \in V$ is given as t_{ij}. Matrix $[t_{ij}]$ satisfies the triangular inequality $t_{ik} \leq t_{ij} + t_{jk}, i, j, k \in V$.

The drones have to perform back-and-forth trips between the depot and the customers' locations to deliver the parcels. Travel times and ranges of drone missions depend on factors such as the number of drones cooperating and the traveling speed. Given a customer i and a number k of drones executing the mission, it is possible to pre-calculate the optimal speed and consequently the total travel time τ_i^k for the back-and-forth trip. When it is not possible to service a customer i for some values of k, then τ_i^k is set to $+\infty$.

We group the customers that can be serviced by truck only in the set $\mathcal{C}_\mathcal{T} \subsetneq \mathcal{C}$. Instead, let $\mathcal{C}_\mathcal{F} = \mathcal{C} \setminus \mathcal{C}_\mathcal{T}$ denote the (sub)set of customers that may be served with some drones' configuration, and let q_j and p_j be the minimum and maximum number of drones to serve a customer $j \in \mathcal{C}_\mathcal{F}$. We adopted the realistic model described in [22] for the calculation of their travel times, and we refer the interested reader to this paper for full details.

Note that a main difficulty of the problem is that once k drones collaborate for a delivery mission, strict synchronization constraints must be fulfilled. The objective of the PDSTSP-c is to find a truck tour, drone-customer assignments, and drones scheduling that minimize the makespan (i.e., the maximum completion time at which all vehicles are back at the depot after completing their services) while fulfilling all the constraints and conditions listed above.

6.1.2 A Constraint Programming model

The Constraint Programming model we present is based on the Google OR-Tools CP-SAT solver [150] and follows the ideas behind the Mixed Integer Linear

Program described in [22]. In particular, drone missions are modeled through a flow. Changes have however been introduced to take full advantage of the characteristics of the solver used.

The CP-SAT solver is designed to work in a multi-thread environment (compatible with all new processors) and can be seen as a portfolio-strategy with some limited data exchange among the different threads. The main process runs a Constraint Programming Solver based on a Lazy Clause Generation (LCG) [151], but other unrelated methods work in parallel to support it and exchange information such as new bounds and solutions. The concept behind LCG involves the (incremental) transformation of the problem into an SAT-formula, subsequently employing an SAT-solver to seek a solution (or prove bounds by infeasibility). The model also gets linearized to some degree, and the corresponding linear program gets (partially) solved with the (dual) simplex algorithm, and other classic MILP techniques are run to enhance bounds and retrieve new solutions, aiming at supporting the satisfiability model. Finally, different instances of a Large Neighborhood Search (LNS) meta-heuristic, seeking for high-quality feasible solutions, are executed.

While the idea above may initially appear as an inefficient approach due to potential redundancy, it proves highly effective in practice. The rationale behind this lies in the inherent challenge of predicting which algorithm is best suited to solve a given problem (No Free Lunch Theorem, [152]). Thus, the pragmatic strategy involves running various approaches in parallel, with the hope that one will effectively address the problem at hand. In contrast, Branch and Cut-based Mixed Integer Programming solvers like Gurobi [153] implement a more efficient partitioning of the search space to reduce redundancy. However, they specialize in a particular strategy, which may not always be the optimal choice.

The variables used in the CP model we present for the PDSTSP-c as follows:

- x_{ij}: binary variables equal to 1 (*true*) if edge (i, j), with $i, j \in V$, is traveled by the truck, 0 (*false*) otherwise. Whereas a loop $x_{jj} = 1$ means that customer j is served by drones, while $x_{jj} = 0$ if it is served by the truck.

- z_j^k: binary variables equal to 1 if customer $j \in \mathcal{C}_\mathcal{F}$ is served by k drones, 0 otherwise.

- y_{ij}: binary variables equal to 1 if vertex i is served right before vertex j within the schedule of any drone, 0 otherwise.

- $f_{ij} \in Z^+$: continuous flow variables indicating number of drones serving vertex i right before vertex j in their schedule.

- $T_j \in R^+$: continuous variables representing the time at which the mission to customer $j \in \mathcal{C}_\mathcal{F}$ is completed by the drones. T_0 is the start time of the operations (typically 0), with all the vehicles at the depot.

- $\alpha \in R^+$: continuous variable denoting the completion time, by which all the carriers are back to the depot.

$$(CP1): \quad \min \alpha \tag{32}$$

$$s.t. \quad \alpha \geq \sum_{i \in V} \sum_{j \in V, i \neq j} t_{ij} x_{ij} \tag{33}$$

$$\alpha \geq T_j \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{34}$$

$$x_{jj} = \sum_{q_j \leq k \leq p_j} z_j^k \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{35}$$

$$\text{Circuit}(x_{ij}, \text{ with } i, j \in V, j \neq i \text{ if } j \in \mathcal{C}_{\mathcal{T}}) \tag{36}$$

$$\sum_{j \in \mathcal{C}_{\mathcal{F}}} f_{0j} \leq m \tag{37}$$

$$\sum_{i \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j} f_{ij} = \sum_{q_j \leq k \leq p_j} k z_j^k \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{38}$$

$$\sum_{i \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j} f_{ij} = \sum_{l \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, l \neq j} f_{jl} \qquad j \in \mathcal{C}_{\mathcal{F}} \cup \{0\} \tag{39}$$

$$f_{ij} \leq m y_{ij} \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{40}$$

$$y_{ij} \implies T_j \geq T_i + \sum_{q_j \leq k \leq p_j} \tau_j^k z_j^k \qquad i \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, j \in \mathcal{C}_{\mathcal{F}}, i \neq j \tag{41}$$

$$0 \leq f_{ij} \leq m \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{42}$$

$$x_{ij} \in \{0; 1\} \qquad i, j \in V \tag{43}$$

$$z_j^k \in \{0; 1\} \qquad j \in \mathcal{C}_{\mathcal{F}}, q_j \leq k \leq p_j \tag{44}$$

$$y_{ij} \in \{0; 1\} \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{45}$$

$$T_j \geq 0 \qquad j \in \mathcal{C}_{\mathcal{F}} \cup \{0\} \tag{46}$$

Following the trivial objective function (32), the constraints have the following meaning. Constraint (33) says that the total time α has to be greater than or equal to the time required by the truck tour. Analogously, constraints (34) impose that α has to be greater than or equal to the completion time of the eventual drone mission to serve customer j. Given the logic of the variables, constraints (35) state that each drone-eligible customer has to be visited either by the truck or by a group of drones; Constraint (36) uses the CP-SAT method *Circuit* [150] to have a feasible truck tour that skips each customer j for which $x_{jj} = 1$ (these customers will be visited by drones). Notice that the input parameter for the *Circuit* command is a set of arcs that the tour can visit, including self-loops. In our case, we exclude only those x_{jj} for which $j \in \mathcal{C}_{\mathcal{T}}$ (corresponding to customers that are not drone-eligible). The Constraints (37)-(39) model the operations and synchronization of the drones as a flow problem (see [22] for more detailed explanations): Constraint (37) states the flow going out from node 0 has to be less than or equal to m (remind that each drone is represented as a unit of flow); Constraints (38) impose that if a customer j is serviced by k drones, than the flow entering node j has to equal k; Constraints (39) are classic conservation equalities, imposing that the flows entering and exiting a node must be equal. Constraints (40) activate the

variables y corresponding to arcs used by flows (variables f) which are necessary to calculate the completion time of drones. Constraints (41) are active only if the variable $y_{ij} = 1$ and state that the synchronization constraint on arc (i, j) must be respected. This is achieved through the CP-SAT command *OnlyEnforceIf* [150], which is indicated with $\implies$ in the model. The remaining constraints (42)-(46) define the domain of the variables.

6.1.3 Valid inequality

The following valid inequality can be introduced to improve the linear relaxation of model *CP1*.

Theorem 1. *The following inequality is valid for the model* CP1*:*

$$m\alpha \geq \sum_{j \in \mathcal{C}_\mathcal{F}} \sum_{q_j \leq k \leq p_j} k\tau_j^k z_j^k \tag{47}$$

Proof. The value of α has to be greater than the maximum completion time of drone missions among all drone-eligible customers (from constraints (34)), which in turn is (by definition) greater than or equal to the average time spent into missions by the drones. This last value is obtained by dividing by the number of drones (m) the cumulative time spent on drone-missions, expressed for each accomplished mission as the time of the mission itself (τ_j^k) multiplied by the number of drones involved (k). Formally:

$$\alpha \geq \max_{j \in \mathcal{C}_\mathcal{F}} T_j \geq \frac{\sum_{j \in \mathcal{C}_\mathcal{F}} \sum_{q_j \leq k \leq p_j} k\tau_j^k z_j^k}{m} \Rightarrow (47)$$

□

The valid inequality (47) will intuitively be tight when the following two conditions hold: (i) the time waited by the drones to synchronize prior to multi-drone missions is small, since this time is not captured by the inequality; (ii) the vehicles (and the drones in particular) have similar total mission times, therefore making the maximum completion time comparable to the average mission time. Notice that both these conditions tend to be fulfilled by optimized solutions.

Finally, we anticipate that the inequality (47) will be valid also for all the models discussed in Section 6.2 for the PDSVRP-c.

6.2 Parallel Drone Scheduling Vehicle Routing Problem with Collective Drones

In this section, we build upon Section 6.1 and introduce the PDSVRP-c, a natural extension of the PDSTSP-c where multiple vehicles operate in parallel to the drones. The problem is introduced in Section 6.2.1 while two models based on Constraint Programming are discussed in Sections 6.2.2 and 6.2.3. The first one is a 2-indices formulation based on the $CP1$ model of the previous section while the second one is a 3-indices formulation. Section 6.2.4 outlines a MILP model to serve as a baseline.

6.2.1 Problem Description

A formal definition of the PDSVRP-c can be proposed as a straightforward extension of the PDSTSP-c provided in Section 6.1.1. The difference is that now we have a fleet S of s trucks, with the same characteristics of the single truck employed for the PDSTSP-c: unlimited capacity, unlimited range, and same traveling speed. No concept of collaboration exists for the trucks and each customer has to be served either by one of the trucks or by drones.

Having a fleet of trucks does not change substantially the problem, but has an impact on the optimization since we now have to plan multiple tours and account for the mission time of each truck while calculating the completion time α. We will see in the next sections two alternative Constraint Programming models and a Mixed Integer Linear Programming formulation.

6.2.2 A 2-indices Constraint Programming model

This model is the direct extension of that discussed in Section 6.1.2 for the $CP1$ and delegates the Constraint Programming solver to handle the multiple truck tours. The variables remain the same, although now the x can take the shape of multiple tours instead of a single one. Another important difference is the definition of the variables T_j. In the $CP1$ model of Section 6.1.2, they are only related to the drones and represent the time in which the mission to a customer is completed. Here, they are extended to the customers served by the trucks and represent the *starting* time of the service of the truck to the customer. Formally we use the new variables $\overline{T}$ with the following meaning

- $\overline{T}_j \in R^+$: continuous variables with a different meaning depending of the type of vehicle involved. It represents the time at which the mission to customer $j \in \mathcal{C}_\mathcal{F}$ is completed when the visit is carried out by drones (it is the time they are back to the depot). If the customer is serviced by a truck, it is the time the truck reaches the customer and the service is started. $\overline{T}_0$ denotes the start time of the operations (typically 0), with all the vehicles at the depot.

$$(CP2) \quad \min \alpha \tag{48}$$

$$s.t. \quad \alpha \geq \overline{T}_j + t_{j0}x_{j0} \qquad j \in \mathcal{C} \tag{49}$$

$$x_{jj} = \sum_{q_j \leq k \leq p_j} z_j^k \qquad j \in \mathcal{C}_\mathcal{F} \tag{50}$$

$$\text{MultipleCircuit}(x_{ij}, \text{ with } i,j \in V, i \neq 0 \vee j \neq 0, j \neq i \text{ if } i \in \mathcal{C}_\mathcal{T}) \tag{51}$$

$$\sum_{j \in \mathcal{C}} x_{0j} \leq s \tag{52}$$

$$x_{ij} \implies \overline{T}_j \geq \overline{T}_i + t_{ij} \qquad i \in V, j \in \mathcal{C}, i \neq j \tag{53}$$

$$\sum_{j \in \mathcal{C}_\mathcal{F}} f_{0j} \leq m \tag{54}$$

$$\sum_{i \in \mathcal{C}_\mathcal{F} \cup \{0\}, i \neq j} f_{ij} = \sum_{q_j \leq k \leq p_j} k z_j^k \qquad j \in \mathcal{C}_\mathcal{F} \tag{55}$$

$$\sum_{i \in \mathcal{C}_\mathcal{F} \cup \{0\}, i \neq j} f_{ij} = \sum_{l \in \mathcal{C}_\mathcal{F} \cup \{0\}, l \neq j} f_{jl} \qquad j \in \mathcal{C}_\mathcal{F} \cup \{0\} \tag{56}$$

$$f_{ij} \leq m y_{ij} \qquad i,j \in \mathcal{C}_\mathcal{F} \cup \{0\}, i \neq j \tag{57}$$

$$y_{ij} \implies \overline{T}_j \geq \overline{T}_i + \sum_{q_j \leq k \leq p_j} \tau_j^k z_j^k \qquad i \in \mathcal{C}_\mathcal{F} \cup \{0\}, j \in \mathcal{C}_\mathcal{F}, i \neq j \tag{58}$$

$$0 \leq f_{ij} \leq m \qquad i,j \in \mathcal{C}_\mathcal{F} \cup \{0\}, i \neq j \tag{59}$$

$$x_{ij} \in \{0;1\} \qquad i,j \in V \tag{60}$$

$$z_j^k \in \{0;1\} \qquad j \in \mathcal{C}_\mathcal{F}, q_j \leq k \leq p_j \tag{61}$$

$$y_{ij} \in \{0;1\} \qquad i,j \in \mathcal{C}_\mathcal{F} \cup \{0\}, i \neq j \tag{62}$$

$$\overline{T}_j \geq 0 \qquad j \in V \tag{63}$$

The constraints strictly follow the meaning already described for the $CP1$ model in Section 6.1.2. The only changes are as follows. Constraints (49) are now extended to cover also the case of truck visits. In this case, α is defined based on the time required by each truck to go back to the depot after visiting each of its assigned customers. This constraint is valid since the travel times satisfy the triangular property, although it could be made valid also for the general case with the use of a *OnlyEnforceIf* statement (see below). Constraint (51) describes a set of circuits through the *MultipleCircuit* command of CP-SAT ([150]) to reflect we are now dealing with several tours instead of one. The command takes in input the set of arcs feasible to be traversed, with the possibility of self-loops in case a customer is visited by the drones. Notice that the arc $(0,0)$ is excluded to avoid subtours not involving the depot, as well as self-loops involving customers that cannot be visited by the drones. The number of tours is not a parameter of the command, and therefore the new constraint (52) is necessary to force the number of tours to be s at most. Whereas the new constraints (53) calculate the service start time for each customer visited

by a truck (remember that $\implies$ indicates the *OnlyEnforceIf*, which activates the constraint if, and only if, $x_{ij} = 1$).

6.2.3 A 3-indices Constraint Programming model

This model is another extension of the $CP1$ model in Section 6.1.2 that uses s separate sets of variables to describe the tours of the s trucks. All the variables remain the same, apart from the x variables which are substituted by a set of variables w such that $w_{ij}^k = 1$ if edge (i, j) is traveled by truck $k \in S$, 0 otherwise. Notice that $w_{jj}^k = 1$ means that customer j is not served by truck k, hence it is not part of its tour. In addition, $w_{00}^k = 1$ means that truck k is not operated in the solution. Notice that differently from model $CP2$, here all the loop variables for the truck are inserted when invoking the method *Circuit* used to find a circuit for each truck (see (69) below) since now only one of the trucks will have to visit a node contained in $\mathcal{C}_\mathcal{T}$ (as imposed by the constraints (68) below). Finally, notice that the timing variables T are the same used in model $CP1$ of Section 6.1.2.

$$(CP3) \quad \min \alpha \tag{64}$$

$$s.t. \quad \alpha \geq \sum_{i \in V} \sum_{j \in V, i \neq j} t_{ij} w_{ij}^k \qquad k \in S \tag{65}$$

$$\alpha \geq T_j \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{66}$$

$$\sum_{k=1}^{s} (1 - w_{jj}^k) + \sum_{q_j \leq k \leq p_j} z_j^k = 1 \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{67}$$

$$\sum_{k=1}^{s} w_{jj}^k = s - 1 \qquad j \in \mathcal{C}_{\mathcal{T}} \tag{68}$$

$$\text{Circuit}(w_{ij}^k, \text{ with } i, j \in V) \qquad k \in S \tag{69}$$

$$w_{ij}^k \leq 1 - w_{00}^k \qquad k \in S, i, j \in \mathcal{C} \tag{70}$$

$$\sum_{j \in \mathcal{C}_{\mathcal{F}}} f_{0j} \leq m \tag{71}$$

$$\sum_{i \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j} f_{ij} = \sum_{q_j \leq k \leq p_j} k z_j^k \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{72}$$

$$\sum_{i \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j} f_{ij} = \sum_{l \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, l \neq j} f_{jl} \qquad j \in \mathcal{C}_{\mathcal{F}} \cup \{0\} \tag{73}$$

$$f_{ij} \leq m y_{ij} \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{74}$$

$$y_{ij} \implies T_j \geq T_i + \sum_{q_j \leq k \leq p_j} \tau_j^k z_j^k \qquad i \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, j \in \mathcal{C}_{\mathcal{F}}, i \neq j \tag{75}$$

$$0 \leq f_{ij} \leq m \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{76}$$

$$w_{ij}^k \in \{0; 1\} \qquad k \in S, i, j \in V \tag{77}$$

$$z_j^k \in \{0; 1\} \qquad j \in \mathcal{C}_{\mathcal{F}}, q_j \leq k \leq p_j \tag{78}$$

$$y_{ij} \in \{0; 1\} \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{79}$$

$$T_j \geq 0 \qquad j \in \mathcal{C}_{\mathcal{F}} \cup \{0\} \tag{80}$$

The constraints strictly follow the meaning already described for the $CP1$ model in Section 6.1.2. The changes reflect the presence of multiple trucks and are as follows. Inequalities (65) now constrain α to be equal to or larger than the length of the tour of each truck k. Equalities (67) now express that each drone-eligible customer has to be visited either by one of the trucks or the drones. The new constraints (68) state that customers in $\mathcal{C}_{\mathcal{T}}$ cannot be visited by drones, and have to be serviced by exactly one truck. Constraints (69) are now independently defined for each truck k, dropping the concept of giant-tour introduced for the $CP2$ model. The new technical constraint (70) forces the circuit of a truck k to be empty once the relative variable w_{00}^k takes the value 1.

6.2.4 A 3-indices Mixed Integer Linear Programming model

We finally present the Mixed Integer Linear Programming formulation of the PDSVRP-c, which is based on the 3-indices $CP3$ model of Section 6.2.3. For the sake of simplicity in the presentation of the model, we adopt a new variable u_j^k that for $j \in C$ takes value 1 if $k \in S$ serves customer j, 0 otherwise. In case $j = 0$, u_0^k takes instead value 1 if $k \in S$ is deployed (used), 0 otherwise. Notice that this variable can be defined as $u_j^k = 1 - w_{jj}^k$ in the logic of the $CP3$ model, but in the MILP model the loop variables w_{jj}^k are not used.

$$(MILP) \quad \min \alpha \tag{81}$$

$$s.t. \quad \alpha \geq \sum_{i \in V} \sum_{j \in V, i \neq j} t_{ij} w_{ij}^k \qquad k \in S \tag{82}$$

$$\alpha \geq T_j \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{83}$$

$$\sum_{k \in S} u_j^k + \sum_{q_j \leq k \leq p_j} z_j^k = 1 \qquad j \in \mathcal{C}_{\mathcal{F}} \tag{84}$$

$$\sum_{k \in S} u_j^k = 1 \qquad j \in \mathcal{C}_{\mathcal{T}} \tag{85}$$

$$u_j^k \leq u_0^k \qquad j \in \mathcal{C}, k \in S \tag{86}$$

$$\sum_{i \in V, i \neq j} w_{ij}^k + \sum_{l \in V, l \neq j} w_{jl}^k = 2u_j^h \qquad j \in V, k \in S \tag{87}$$

$$\sum_{i,j \in H, i \neq j} w_{ij}^k \leq |H| - 1 \qquad H \subseteq \mathcal{C}, k \in S \tag{88}$$

$$\frac{f_{ij}}{m} \leq y_{ij} \leq f_{ij} \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{89}$$

$$T_j + M(1 - y_{ij}) \geq T_i + \sum_{q_j \leq k \leq p_j} \tau_j^k z_j^k \quad i \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, j \in \mathcal{C}_{\mathcal{F}}, i \neq j \tag{90}$$

$$0 \leq f_{ij} \leq m \qquad i, j \in \mathcal{C}_{\mathcal{F}}, i \neq j \tag{91}$$

$$w_{ij}^k \in \{0; 1\} \qquad k \in S, i, j \in V, i \neq j \tag{92}$$

$$z_j^k \in \{0; 1\} \qquad j \in \mathcal{C}_{\mathcal{F}}, q_j \leq k \leq p_j \tag{93}$$

$$u_j^k \in \{0; 1\} \qquad j \in \mathcal{C}, k \in S \tag{94}$$

$$y_{ij} \in \{0; 1\} \qquad i, j \in \mathcal{C}_{\mathcal{F}} \cup \{0\}, i \neq j \tag{95}$$

$$T_j \geq 0 \qquad j \in \mathcal{C}_{\mathcal{F}} \cup \{0\} \tag{96}$$

The model minimizes the time to serve all the customers (81). Constraints (82) force α to be larger than any tour of the trucks, and inequalities (83) guarantee that α is larger than the completion time of any drone's mission time. Equations (84) assign customers from $\mathcal{C}_{\mathcal{F}}$ to either a drone or a truck, while constraints (85) force the customers that can be visited only by a truck ($\mathcal{C}_{\mathcal{T}}$) to receive such a visit. Inequalities (86) impose that a customer can be visited by

a truck only if it is in use. Equalities (87) are flow conservation constraints for the truck tours. Inequalities are subtour elimination constraints [154] (further defined below) and guarantee that truck tours are circuits including the depot. Notice that these constraints are exponential in number, depending on any possible subset H of $\mathcal{C}$. In our implementation, they will be generated dynamically as described below. Constraints (89) refers to the flow of drones and guarantee that the number of drones going from customer i to j must be lower than m, only if there is a flow from i to j. Inequalities (90) regulate completion of the service time for the customers visited by the drones. Finally, constraints (91)-(96) define the domain of the variables.

Separation of the subtour elimination constraints

The subtour elimination constraints (88) are dynamically added to the MILP as *Lazy constraints* (available in the most popular MILP solvers). Specifically, the solver starts by disregarding the constraints declared *lazy*, and once a feasible integer solution is found, it invokes a user-defined separation procedure. In our case, since the solution at hand is integer, the separation is a simple $O(|E|)$ exploration of the graphs $G^k = (V, E^k)$ with $E^k = \{(i, j) \in E : w_{ij}^k = 1\}$ to look for subtours not involving the depot.

6.3 Results

All the models presented in previous sections have been coded in Python 3.11.2. The Constraint Programming models of Sections 6.1 and 6.2 have been solved via the CP-SAT solver of Google OR-Tools 9.6 [150] while the Mixed Integer Linear model of Section 6.2 has been solved with Gurobi 10.0 [153].

The outcome of the experimental campaign is discussed in the remainder of this section and is organized according to the tackled problems. Tables 17-24 report, for each instance: i) the instance name; ii) the lower bound eventually produced and the best heuristic solution ([LB, UB]); iii) the computing time to find the best heuristic solution (Sec_{bst}); iv) the eventual computing time to prove optimality (Sec_{tot}); v) a final summary column (Best bounds) containing the current state-of-the-art results of each instance for easing future research.

In addition, we use a dash whenever a result is not retrieved or the time limit is reached, and we mark in *italics* the results of our models not matching nor improving best-known bounds while in **bold** those producing new best bounds. A line with the average of the relevant column is present in the bottom of each table, to ease the interpretation of the results. Hardware configurations, solvers used, experimental settings and time limits are finally reported in the notes of the tables for each approach.

6.3.1 Benchmark Instances

To evaluate the performance of the proposed models for both the PDSTSP-c and the new PDSVRP-c, we consider the instances originally introduced in [22]

for the PDSTSP-c[2]. The number n of customers varies from 15 to 200 (first number of the instance name) and the instances are divided into small ($n \leq 30$) and large ($n > 30$). The number m of drones available varies in the range $[3, 6]$ for the small instances and $[5, 10]$ for the large ones. The traveling distances for trucks are computed using Manhattan distances and a speed of 30 km/h, while drones follow the Euclidean distance and the optimal travel times (rounded up to the nearest integer) are pre-calculated for each collaborative cluster of k drones. The interested reader can find all the details of the instances in [22].

For generating PDSVRP-c instances, we used the same set of benchmarks and added the number s of trucks chosen in the range $[2, 3]$ for small instances and in $[2, 5]$ for large instances.

6.3.2 PDSTSP-c

In this section, we compare the results obtained by solving the $CP1$ model of Section 6.1.2, with and without the valid inequality (47). The results are summarized in Table 17 for the small instances and in Table 18 for the large ones. We compare $CP1$ with the methods introduced in [22], namely a MILP model solved with IBM CPLEX 12.1 [155] and two versions of a *Ruin & Recreate* metaheuristic: RnR fast and RnR. Notice that the results of the MILP model in [22] are only available for small instances and those reported for the *Ruin&Recreate* methods are the best over 30 runs. To fully understand the impact of inequality (47), we also considered the MILP model described in Section 6.2.4 for the PDSVRP-c, and run it with $s = 1$ (one truck only) as well as the inequality (47). This method is run on small instances only, since [22]) demonstrated that MILP models are not suitable for large instances.

We are not aware of other existing methods to deal with this problem.

[2]Available at: http://orlab.com.vn/home/download

Table 17: Experimental results on the PDSTSP-c. Small instances.

Instance	RnR fast [22][a]		RnR [22][a]		MILP [22][b]		*MILP*+(47)[c]			*CP*1[d]			*CP*1+ (47)[d]			Best
	UB	Sec_{bst}	UB	Sec_{bst}	[LB, UB]	Sec_{tot}	[LB, UB]	Sec_{tot}	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	bounds
15-r-e	92	0.32	92	0.95	[92, 92]	8.65	[92, 92]	1215.79	850.00	[92, 92]	0.84	0.44	[92, 92]	1.14	1.10	92
15-rc-c	44	0.43	44	1.46	[31.74, 44]	-	[**44**, 44]	1837.95	1790.09	[40, 44]	-	28.79	[**44**, 44]	12.00	11.94	44
16-c-c	60	0.52	60	2.14	[60, 60]	2.61	[**60**, 60]	3.66	3.63	[**60**, 60]	2.95	2.90	[60, 60]	2.10	2.05	60
16-r-e	112	0.55	112	1.52	[112, 112]	63.86	[98.20, 112]	-	250.10	[112, 112]	2.00	1.94	[112, 112]	1.61	1.57	112
18-c-c	56	0.42	56	1.86	[38.72, 56]	-	[**56**, 56]	1258.06	50.89	[44, 56]	-	302.76	[**56**, 56]	42.22	42.13	56
18-r-e	96	0.59	96	1.79	[86.94, 96]	-	[87.86, 96]	-	2338.15	[92, 96]	-	55.83	[**96**, 96]	4.70	4.63	96
18-rc-c	58	0.54	57	2.40	[37.78, 57]	-	[56, 57]	-	3060.21	[38, *60*]	-	399.43	[**57**, 57]	1803.93	6.57	57
19-c-c	44	0.48	44	1.81	[28.06, 44]	-	[40.85, 44]	-	3302.58	[32, 44]	-	64.50	[**44**, 44]	24.91	11.00	44
20-c-c	43	0.60	43	2.54	[30.99, 44]	-	[39.42, 43]	-	2374.04	[**40**, 43]	-	102.82	[**40**, 43]	-	147.26	[40, 43]
20-r-c	64	0.43	64	1.91	[55.35, 64]	-	[61.80, 64]	-	3326.72	[56, 64]	-	276.15	[**64**, 64]	73.39	73.29	64
20-r-e	82	0.62	80	2.00	[62.59, 88]	-	[72.80, 82]	-	3237.98	[72, 80]	-	606.13	[**80**, 80]	38.11	38.00	80
20-rc-c	96	0.41	96	2.37	[96, 96]	0.9	[96, 96]	460.30	1.14	[96, 96]	6.48	6.42	[96, 96]	6.23	6.17	96
20-rc-e	100	0.46	100	1.09	[100, 100]	88.78	[90, 100]	-	292.98	[100, 100]	6.92	6.87	[100, 100]	4.70	4.65	100
21-c-c	62	0.48	62	2.25	[41.80, 64]	-	[44, 64]	-	2281.04	[*36*, *64*]	-	18.02	[**60**, 62]	-	58.44	[60, 62]
21-r-e	85	0.59	85	1.74	[59.52, 100]	-	[75.25, 88]	-	3572.98	[*49*, *88*]	-	1512.11	[**85**, 85]	940.55	128.94	85
23-c-e	80	0.60	80	2.49	[58.15, 80]	-	[58.15, 80]	-	2698.31	[**80**, 80]	0.84	0.78	[**80**, 80]	1.31	1.25	80
23-r-c	88	0.42	88	1.83	[88, 88]	3293.16	[84, 88]	-	3042.55	[88, 88]	218.07	217.97	[88, 88]	8.97	8.90	88
24-c-e	84	0.79	84	2.50	[78.4, 84]	-	[78.40, 84]	-	3567.08	[**84**, 84]	14.07	13.98	[**84**, 84]	11.05	10.97	84
24-r-e	112	0.52	112	1.57	[91.05, 112]	-	[101, 112]	-	1819.74	[108, 112]	-	4.76	[**112**, 112]	955.94	3.60	112
24-rc-c	72	0.73	71	4.06	[69.58, 88]	-	[69.58, 72]	-	14.13	[*68*, 71]	-	2245.17	[**70**, **70**]	190.03	189.86	70
25-c-c	56	0.60	56	2.92	[37.33, 56]	-	[37.83, 56]	-	694.26	[*35*, 56]	-	764.50	[**56**, 56]	44.85	38.62	56
25-r-e	106	0.96	104	3.14	[76.11, 120]	-	[95.73, 108]	-	3533.88	[*58*, *108*]	-	60.58	[**104**, 104]	288.19	288.04	104
25-rc-e	92	0.71	92	2.25	[66.99, 100]	-	[83.60, 96]	-	2636.83	[*60*, *97*]	-	90.45	[**92**, 92]	113.76	113.63	92
26-r-c	103	0.53	103	2.58	[95.26, 128]	-	[100.18, 103]	-	3409.09	[*84*, *104*]	-	2746.86	[**101**, 103]	-	107.06	[101, 103]
27-c-c	84	0.49	84	2.18	[83.23, 84]	-	[64.72, 84]	-	2672.84	[84, 84]	1.91	1.85	[**84**, 84]	1.80	1.75	84
27-c-e	68	0.72	68	6.27	[42.04, 68]	-	[42.04, 68]	-	1429.86	[*31*, 68]	-	1.76	[**68**, 68]	33.23	1.36	68
27-rc-c	100	0.77	100	5.30	[100, 100]	721.25	[85.34, 100]	-	2915.17	[100, 100]	394.35	394.19	[100, 100]	71.17	71.06	100
27-rc-e	84	0.79	84	2.70	[59.52, 100]	-	[64.29, 88]	-	1202.70	[*42*, *88*]	-	2066.04	[**84**, 84]	576.56	54.8	84
29-rc-e	116	0.72	116	1.69	[97.71, 124]	-	[109.75, 116]	-	2214.71	[**116**, 116]	654.58	654.40	[**116**, 116]	8.60	8.53	116
30-c-c	96	0.65	96	3.76	[83.78, 96]	-	[83.78, 96]	-	1827.64	[**96**, 96]	2.30	2.22	[**96**, 96]	3.22	3.14	96
Average	81.17	0.58	80.97	2.44	[68.69, 84.83]	-	[72.42, 81.63]	-	2013.71	[69.77, 81.70]	-	421.69	[80.70, 80.93]	-	48.01	[80.70, 80.93]

[a] CPU AMD Ryzen 3700X - 4x3.6 GHz, 4x4.4 GHz, 16 threads; RAM 32 GB; best results over 30 runs
[b] CPU AMD Ryzen 3700X - 4x3.6 GHz, 4x4.4 GHz, 16 threads; RAM 32 GB; CPLEX 12.1; 3600 sec time limit
[c] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; Gurobi 10.0; 3600 sec time limit
[d] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

From the results displayed in Table 17, we see that inequalities (47) are very effective in improving the performance of the models, both $CP1$ and $MILP$. Given this, we will always consider these inequalities for the next experiments.

Table 17 reveals that the CP approach matches (or improves in the case of instance 24-rc-c) all the best-known heuristic solutions and outperforms the MILP method, both in terms of quality and times. Additionally, we observe how the $CP1$+(47) improves several lower bounds and closes all but three instances.

To better highlight the differences between the MILP and the CP methods, we report in Figure 15 their percentage optimality gaps, calculated as $100 \cdot \frac{UB-LB}{UB}$, and in Figure 16 their required time to find the best solution (Sec_{bst}). Figure 15 shows that the $CP1$ model clearly leads to lower optimality gaps than the $MILP$ model with a time limit of 3600 seconds, the latter also demonstrating scalability issues on larger instances as remarked by its linearly increasing trend (dashed line). Whereas Figure 16 shows that the $CP1$ model is substantially faster in retrieving the best heuristic solution. These results suggest that the Constraint Programming-based approach has great potential for the PDSTSP-c problem.

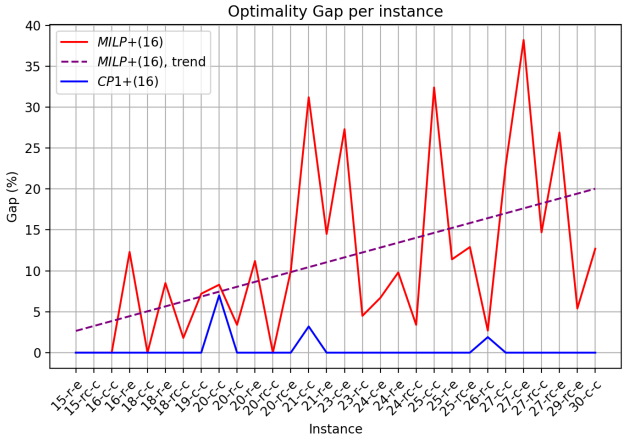

Figure 15: The optimality gap in percentage for the $MILP$+(47) and $CP1$+(47) on small PDSTSP-c instances.

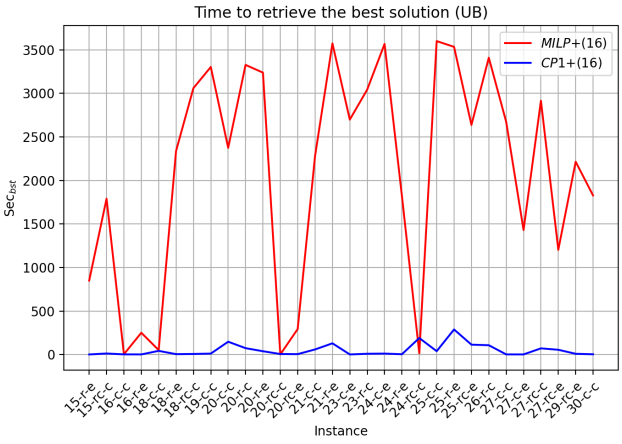

Figure 16: The time required in seconds by the $MILP$+(47) and $CP1$+(47) to retrieve the best heuristic solution (UB).

Table 18: Experimental results on the PDSTSP-c. Large instances.

	RnR fast [22][a]		RnR [22][a]		$CP1$[b]			$CP1$+(47)[b]			Best
Instance	UB	Sec_{bst}	UB	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	bounds
50-r-e	120	2.45	116	20.00	[49, *140*]	-	2123.27	[**116** , *128*]	-	2083.72	116
53-r-e	136	2.23	132	15.17	[68, *160*]	-	2631.50	[**132**, *140*]	-	3177.59	132
66-rc-e	128	3.13	124	16.61	[80, *168*]	-	393.29	[**124**, *132*]	-	2927.03	124
67-c-c	76	2.32	76	29.05	[68, *80*]	-	133.82	[**73**, *80*]	-	403.51	[73, 76]
68-rc-c	79	2.12	76	20.45	[72, *112*]	-	2488.01	[**76**, 76]	3089.06	3088.37	76
76-c-c	52	2.15	52	10.46	[40, 52]	-	26.87	[**52**, 52]	379.86	31.99	52
82-c-e	64	2.63	64	18.07	[**64**, 64]	33.30	30.55	[**64**, 64]	59.21	58.42	64
82-rc-c	108	2.65	104	27.42	[100, *140*]	-	2373.65	[**104**, *144*]	-	3269.15	104
88-c-e	108	3.56	108	9.62	[**108**, 108]	51.03	50.09	[**108**, 108]	92.47	91.56	108
91-r-c	128	3.64	124	72.49	[116, *188*]	-	2887.23	[**120**, *164*]	-	3302.76	[120, 124]
99-rc-e	108	3.64	100	51.21	[80, *168*]	-	1640.06	[**98**, *164*]	-	2575.79	[98, 100]
101-r-c	124	3.98	120	99.71	[96, *176*]	-	3456.53	[**114**, *180*]	-	3338.22	[114, 120]
103-rc-c	128	4.38	124	94.25	[108, *176*]	-	1967.67	[**120**, *164*]	-	2147.63	[120, 124]
105-rc-e	124	5.78	120	62.71	[80, *184*]	-	903.40	[**109**, *132*]	-	1831.25	[109, 120]
108-rc-e	144	5.56	136	61.71	[112, *188*]	-	3088.15	[**134**, *188*]	-	2521.07	[134, 136]
114-rc-c	100	4.69	96	62.73	[68, *152*]	-	2591.56	[**94**, *148*]	-	2821.57	[94, 96]
121-rc-e	128	6.41	124	68.74	[108, *180*]	-	2865.31	[**121**, *160*]	-	1201.60	[121, 124]
126-r-c	161	5.89	160	120.93	[104, *228*]	-	1457.72	[**151**, *216*]	-	3028.00	[151, 160]
126-rc-e	148	7.38	144	71.04	[124, *196*]	-	2945.63	[**136**, *188*]	-	1053.63	[136, 144]
144-rc-r	132	6.52	128	172.75	[120, *188*]	-	2801.00	[**122**, *204*]	-	3289.31	[122, 128]
154-c-c	72	7.17	72	62.96	[**68**, 72]	-	63.64	[**68**, 72]	-	61.68	[68, 72]
165-r-c	176	7.61	164	280.18	[118, *292*]	-	2386.33	[**140**, *312*]	-	3528.39	[140, 164]
167-r-e	200	10.55	188	228.31	[72, *296*]	-	3309.32	[**160**, *304*]	-	2209.10	[160, 188]
173-r-c	180	8.64	164	373.43	[92, *312*]	-	2439.90	[**141**, *280*]	-	2184.87	[141, 164]
173-rc-r	144	9.22	133	135.20	[51, *208*]	-	2539.43	[**115**, *208*]	-	3556.41	[115, 133]
181-r-e	232	11.20	224	196.09	[125, *332*]	-	3252.35	[**199**, *348*]	-	3566.96	[199, 224]
185-c-c	96	11.26	96	61.53	[**96**, 96]	1279.96	1276.68	[**96**, 96]	622.29	619.07	96
187-rc-e	200	12.67	196	119.95	[78, *284*]	-	2464.47	[**167**, *288*]	-	3228.65	[167, 196]
198-c-c	64	11.38	64	94.12	[**64**, *68*]	-	82.40	[**64**, *68*]	-	155.40	64
200-r-e	224	13.88	212	368.97	[40, *324*]	-	3541.42	[**162**, *328*]	-	2132.93	[162, 212]
Average	129.47	6.16	124.70	100.86	[85.63, 177.73]	-	1940.38	[116.00, 171.20]	-	2116.19	[116.00, 124.70]

[a] CPU AMD Ryzen 3700X - 4x3.6 GHz, 4x4.4 GHz, 16 threads; RAM 32 GB; best results over 30 runs
[b] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

Moving to the larger instances reported in Table 18, we observe that $CP1$+(47) can provide, for the first time, valid lower bounds for all instances. Furthermore, 10 over 30 bounds equal the best-known solutions, hence proving for the first time the optimality of these solutions. In the remaining instances, the gaps between the lower bound and the heuristic solution are generally small. However, the upper bound provided by the CP models is not competitive with respect to that of the meta-heuristic methods. Also, the running times are larger, although it is worth observing once again that for the *RnR* methods, the best results over 30 runs are provided, making the timing presented less fair.

6.3.3 PDSVRP-c

In this section, we compare the performance of the $CP2$, $CP3$, and $MILP$ models described in Sections 6.2.2-6.2.4 for the PDSVRP-c. Their results are summarized in Tables 19 and 20 for the small instances, covering respectively 2 and 3 trucks, and in Tables 21-24 for the large instances, using respectively 2, 3, 4, and 5 trucks. The PDSVRP-c is first introduced in this work, so no comparison is available with methods from other authors.

Table 19: Experimental results on the PDSVRP-c. Small instances, 2 trucks.

	$MILP$+(47)[a]			$CP2$+(47)[b]			$CP3$+(47)[b]			Best
Instance	[LB, UB]	Sec_{tot}	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	bounds
15-r-e	[**92** , **92**]	2654.11	745.82	[**92**, **92**]	157.1	0.21	[**92**, **92**]	2.49	0.45	92
15-rc-c	[**33**, **33**]	9.42	7.46	[**33**, **33**]	0.92	0.52	[**33**, **33**]	5.14	2.92	33
16-c-c	[**40**, **40**]	50.04	3.42	[**40**, **40**]	6.43	0.29	[**40**, **40**]	7.17	2.26	40
16-r-e	[104, **108**]	-	783.12	[**108**, **108**]	83.49	0.69	[**108**, **108**]	5.03	1.75	108
18-c-c	[**44**, **44**]	22.76	22.72	[**44**, **44**]	3.09	2.4	[**44**, **44**]	7.45	5.98	44
18-r-e	[**92**, **92**]	1153.19	197.42	[**92**, **92**]	43.36	1.57	[**92**, **92**]	6.92	4.52	92
18-rc-c	[44, **46**]	-	3569.88	[**46**, **46**]	459.47	161.29	[**46**, **46**]	401.32	45.46	46
19-c-c	[34, **36**]	-	3220.87	[**36**, **36**]	7.85	4.15	[**36**, **36**]	15.41	3.84	36
20-c-c	[**40**, **40**]	907.04	5.25	[**40**, **40**]	4.35	2.23	[**40**, **40**]	4.26	1.75	40
20-r-c	[**48**, **48**]	2023.70	1886.70	[**48**, **48**]	454.46	9.19	[**48**, **48**]	62.72	43.56	48
20-r-e	[63, 76]	-	2707.00	[**72**, **72**]	462.23	331.95	[**72**, **72**]	27.42	23.27	72
20-rc-c	[63, **64**]	-	1977.44	[58, **64**]	-	1.15	[**64**, **64**]	14.57	8.09	64
20-rc-e	[72, **80**]	-	3031.82	[**64**, **80**]	-	1.49	[**80**, **80**]	24.93	7.82	80
21-c-c	[**40**, **40**]	59.12	8.87	[**40**, **40**]	9.94	1.36	[**40**, **40**]	11.6	2.01	40
21-r-e	[51, **76**]	-	3348.26	[**76**, **76**]	475.63	4.08	[**76**, **76**]	82.46	9.41	76
23-c-e	[44, **80**]	-	0.98	[42, **80**]	-	0.98	[**80**, **80**]	17.85	0.67	80
23-r-c	[**60**, **60**]	1536.90	1379.17	[57, **60**]	-	9.64	[**60**, **60**]	475.01	10.69	60
24-c-e	[56, **60**]	-	505.00	[**60**, **60**]	116.75	34.53	[**60**, **60**]	54.04	53.77	60
24-r-e	[68, **100**]	-	0.89	[67, **100**]	-	3.09	[**100**, **100**]	68.82	35.52	100
24-rc-c	[49, **52**]	-	42.34	[**52**, **52**]	1350.35	25.78	[**52**, **52**]	307.8	82.07	52
25-c-c	[**40**, **40**]	2523.46	1988.78	[**40**, **40**]	899.8	26.8	[**40**, **40**]	79.98	63.91	40
25-r-e	[72, 92]	-	693.55	[85, **88**]	-	35.41	[**88**, **88**]	175.75	44.11	88
25-rc-e	[64, 80]	-	239.90	[59, **76**]	-	80.16	[**76**, **76**]	189.81	3.42	76
26-r-c	[68, **70**]	-	1381.35	[65, **70**]	-	20.87	[**70**, **70**]	2701.81	700.35	70
27-c-c	[40, **52**]	-	83.87	[42, **52**]	-	2.32	[**52**, **52**]	37.07	16.67	52
27-c-e	[7, **68**]	-	6.18	[**68**, **68**]	2741.9	0.65	[**68**, **68**]	130.71	1.56	68
27-rc-c	[64, **72**]	-	58.13	[**64**, **72**]	-	8.96	[**72**, **72**]	79.95	53.96	72
27-rc-e	[36, 80]	-	171.86	[47, **76**]	-	20.22	[**76**, **76**]	198.05	46.71	76
29-rc-e	[62, **100**]	-	3092.20	[65, **100**]	-	13.71	[**100**, **100**]	59.26	48.57	100
30-c-c	[36, **64**]	-	2935.06	[48, **64**]	-	2.09	[**64**, **64**]	39.64	2.99	64
Average	[54.20, 66.17]	-	1136.51	[58.33, 65.63]	-	26.93	[65.63, 65.63]	176.48	44.27	65.63

[a] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; Gurobi 10.0; 3600 sec time limit
[b] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

Table 20: Experimental results on the PDSVRP-c. Small instances, 3 trucks.

	$MILP$+(47)[a]			$CP2$+(47)[b]			$CP3$+(47)[b]			Best
Instance	[LB, UB]	Sec_{tot}	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	[LB, UB]	Sec_{tot}	Sec_{bst}	bounds
15-r-e	[72, **92**]	-	3401.20	[**92**, **92**]	331.05	0.19	[**92**, **92**]	4.44	0.36	92
15-rc-c	[**32**, **32**]	11.08	11.01	[**32**, **32**]	1.43	1.27	[**32**, **32**]	5.95	3.66	32
16-c-c	[**36**, **36**]	2.20	2.18	[**36**, **36**]	1.21	1.17	[**36**, **36**]	6.01	5.41	36
16-r-e	[68, **108**]	-	2332.97	[**108**, **108**]	290.7	1.05	[**108**, **108**]	6.55	1.15	108
18-c-c	[**44**, **44**]	25.27	25.21	[**44**, **44**]	3.02	1.61	[**44**, **44**]	6.37	2.12	44
18-r-e	[88, **92**]	-	614.09	[**92**, **92**]	22.3	0.92	[**92**, **92**]	12	2.1	92
18-rc-c	[**40**, **40**]	18.60	18.53	[**40**, **40**]	5.22	5.09	[**40**, **40**]	30.65	24.69	40
19-c-c	[29, **36**]	-	3554.00	[**36**, **36**]	2.26	0.86	[**36**, **36**]	19.25	3.59	36
20-c-c	[**40**, **40**]	383.19	139.78	[**40**, **40**]	1.9	0.72	[**40**, **40**]	7.35	1.27	40
20-r-c	[**37**, **37**]	1202.72	582.24	[**37**, **37**]	8.3	3.81	[**37**, **37**]	99.17	36.74	37
20-r-e	[44, **72**]	-	45.40	[**72**, **72**]	443.37	7.4	[**72**, **72**]	17.12	9.27	72
20-rc-c	[**48**, **48**]	604.42	8.91	[**48**, **48**]	99.53	3.15	[**48**, **48**]	94.88	26.53	48
20-rc-e	[60, **68**]	-	595.63	[**68**, **68**]	122.18	42	[**68**, **68**]	45.74	8.25	68
21-c-c	[**36**, **36**]	28.87	6.92	[**36**, **36**]	4.96	4.75	[**36**, **36**]	16.5	7.81	36
21-r-e	[34, **76**]	-	689.44	[**76**, **76**]	891.56	11.3	[**76**, **76**]	427.99	28.05	76
23-c-e	[36, **80**]	-	3.88	[66, **80**]	-	0.94	[**80**, **80**]	25.44	0.93	80
23-r-c	[**48**, **48**]	203.22	44.94	[**48**, **48**]	20.92	14.24	[**48**, **48**]	590.1	135.03	48
24-c-e	[56, **60**]	-	2000.96	[**60**, **60**]	135.61	3.35	[**60**, **60**]	28.56	11.45	60
24-r-e	[47, **100**]	-	8.28	[80, **100**]	-	1.72	[**100**, **100**]	64.88	8.77	100
24-rc-c	[41, **44**]	-	3456.94	[**44**, **44**]	227.22	77.94	[**44**, **44**]	926.73	387.43	44
25-c-c	[30, 40]	-	97.39	[**37**, **37**]	70.58	31.83	[**37**, **37**]	107.3	42.94	37
25-r-e	[57, 96]	-	3298.63	[59, **85**]	-	44.6	[**85**, **85**]	404.43	242.54	85
25-rc-e	[52, 69]	-	2601.65	[65, **66**]	-	141.25	[**66**, **66**]	356.83	61.26	66
26-r-c	[**56**, **56**]	180.88	67.14	[52, **56**]	-	1399.21	[55, **56**]	-	70.15	56
27-c-c	[**36**, **36**]	1367.86	936.48	[**36**, **36**]	577.74	3.69	[**36**, **36**]	121.7	39.48	36
27-c-e	[8, **68**]	-	4.92	[**68**, **68**]	2315.01	1.31	[**68**, **68**]	437.75	1.71	68
27-rc-c	[**60**, **60**]	223.20	213.85	[**60**, **60**]	6.79	6.01	[**60**, **60**]	74.79	63.9	60
27-rc-e	[28, **76**]	-	155.68	[56, **76**]	-	8.02	[**76**, **76**]	520.4	14.61	76
29-rc-e	[53, 108]	-	3337.59	[72, **100**]	-	23.67	[**100**, **100**]	56.98	35.42	100
30-c-c	[26, **38**]	-	3182.76	[**38**, **38**]	96.12	5.63	[**38**, **38**]	145.11	15.99	38
Average	[44.73, 61.20]		1047.95	[56.60, 60.37]	-	61.62	[60.33, 60.37]	-	43.09	60.37

[a] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; Gurobi 10.0; 3600 sec time limit
[b] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

Tables 19 and 20 suggest that solving the $MILP$ is less effective than solving the CP models, both in terms of bounds provided and computing time. The only remarkable exception is instance 26-r-c with 3 trucks (Table 20), which is closed by the former but not by the latters.

One can also observe that the performances of $MILP$ degrade with the increasing of the instance size, much more than that of the CP methods. After some tests with larger instances (not reported here), and considering the analogous decision made for the PDSTSP-c in [22], we decided to not consider the $MILP$ for the experiments on large instances (Tables 21-24).

The results of the two CP models suggest that the 3-indices formulation ($CP3$) is superior, being able to close all the instances but one. The 2-indices model appears slower even though the quality of its upper bounds is the same of $CP3$. This highlights that the weakness of the $CP2$ model is in the computation of the lower bound.

The results reported in Tables 21-24 for large instances (notice that the column Sec_{tot} has been omitted, since no optimality is proven) and a varying number of trucks lead to the following observations. The model with 3 indices,

which performs the best on small instances (see Tables 19 and 20), is instead performing worse than the 2-indices model on large ones, especially in terms of retrieved lower bounds. This might suggest that handling multiple truck tours with the *MultipleCircuit* command becomes effective when tours are complex.

There are however a few exceptions where the 3-indices model is better either in terms of lower or upper bounds. Specifically, the $CP3$ model appears to be more consistent in instances with many customers and a few trucks, in which the 2-indices model often fails to produce any feasible solution. This might indicate that the models are approaching their natural limit.

Table 21: Experimental results on the PDSVRP-c. Large instances, 2 trucks.

	$CP2$+(47)[c]		$CP3$+(47)[c]		Best
Instance	[LB, UB]	Sec_{bst}	[LB, UB]	Sec_{bst}	bounds
50-r-e	[**65**, **116**]	206.57	[63, 120]	168.48	[65, 116]
53-r-e	[77, **112**]	894.09	[**82**, 128]	1756.80	[82, 112]
66-rc-e	[72, **112**]	1829.73	[**73**, 136]	866.28	[73, 112]
67-c-c	[**38**, **52**]	22.33	[31, **52**]	827.01	[38, 52]
68-rc-c	[50, **56**]	3332.51	[**52**, 104]	3088.50	[52, 56]
76-c-c	[**26**, **36**]	20.60	[16, 40]	185.95	[26, 36]
82-c-e	[**32**, **64**]	25.41	[17, **64**]	73.68	[32, 64]
82-rc-c	[**62**, **116**]	2974.84	[56, 132]	2615.62	[62, 116]
88-c-e	[54, **84**]	298.18	[**58**, 112]	49.28	[58, 84]
91-r-c	[**75**, **152**]	405.02	[**75**, 160]	2249.67	[75, 152]
99-rc-c	[**63**, **96**]	2083.65	[51, 144]	564.95	[63, 96]
101-rc	[**71**, 164]	2921.49	[53, **152**]	1731.45	[71, 152]
103-rc-c	[**69**, **124**]	2603.95	[52, 128]	2912.93	[69, 124]
105-rc-e	[**65**, **136**]	2170.84	[57, 148]	1383.74	[65, 136]
108-rc-e	[**79**, 172]	1683.07	[70, **160**]	831.13	[79, 160]
114-rc-c	[**58**, **124**]	3417.23	[49, 140]	411.62	[58, 124]
121-rc-e	[**70**, 156]	647.12	[56, **152**]	2088.27	[70, 152]
126-rc-e	[**87**, 220]	3115.59	[67, **184**]	1956.96	[87, 184]
126-r-c	[**78**, 160]	2679.11	[56, **156**]	1448.65	[78, 156]
144-rc-c	[**67**, 272]	2610.83	[47, **168**]	3103.46	[67, 168]
154-c-c	[**35**, -]	-	[8, **72**]	279.16	[35, 72]
165-r-c	[**88**, -]	-	[67, **224**]	3544.74	[88, 224]
167-r-e	[**100**, -]	-	[74, **256**]	3151.22	[100, 256]
173-r-c	[**85**, **204**]	2929.34	[59, 240]	2251.40	[85, 204]
173-rc-c	[**79**, -]	-	[48, **180**]	1797.98	[79, 180]
181-r-e	[**112**, -]	-	[78, **252**]	3388.32	[112, 252]
185-c-c	[**48**, -]	-	[24, **96**]	316.31	[48, 96]
187-rc-e	[**100**, 308]	3391.71	[65, **212**]	1567.38	[100, 212]
198-c-c	[**32**, -]	-	[12, **64**]	271.52	[32, 64]
200-r-e	[**105**, -]	-	[68, **324**]	2072.94	[105, 324]
Average	[68.07, -]	-	[52.80, 150.00]	1565.18	[68.47, 141.20]

[c] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

Table 22: Experimental results on the PDSVRP-c. Large instances, 3 trucks.

Instance	$CP2$+(47)[c] [LB, UB]	Sec_{bst}	$CP3$+(47)[c] [LB, UB]	Sec_{bst}	Best bounds
50-r-e	[**48**, **112**]	79.65	[47, 112]	411.11	[48, 112]
53-r-e	[**56**, **96**]	860.00	[51, 112]	2074.85	[56, 96]
66-rc-e	[**53**, **108**]	282.64	[38, 116]	139.49	[53, 108]
67-c-c	[**27**, **52**]	32.82	[9, **52**]	353.35	[27, 52]
68-rc-c	[**39**, **56**]	756.18	[34, 104]	655.52	[39, 56]
76-c-c	[**18**, **24**]	42.28	[12, 52]	81.65	[18, 24]
82-c-e	[**22**, **64**]	21.88	[8, **64**]	26.79	[22, 64]
82-rc-c	[**47**, **80**]	1727.16	[38, 128]	312.65	[47, 80]
88-c-e	[**36**, **76**]	375.27	[32, 104]	587.30	[36, 76]
91-r-c	[**56**, **120**]	3036.11	[42, 148]	726.56	[56, 120]
99-rc-c	[**47**, **64**]	2650.32	[29, 128]	196.67	[47, 64]
101-rc	[**52**, **128**]	2645.43	[36, 144]	2520.98	[52, 128]
103-rc-c	[**49**, **96**]	2229.84	[32, 136]	2332.89	[49, 96]
105-rc-e	[**49**, **120**]	877.50	[34, 132]	907.44	[49, 120]
108-rc-e	[**58**, 184]	1969.45	[37, **160**]	1273.20	[58, 160]
114-rc-c	[**44**, **80**]	1676.32	[32, 112]	466.45	[44, 80]
121-rc-e	[**52**, **124**]	2820.31	[40, 152]	1701.91	[52, 124]
126-rc-e	[**63**, **136**]	2839.24	[44, 164]	2663.93	[63, 136]
126-r-c	[**56**, **140**]	2191.71	[38, 148]	3114.44	[56, 140]
144-rc-c	[**50**, **132**]	3362.32	[35, 160]	2396.79	[50, 132]
154-c-c	[**24**, **36**]	195.44	[8, 68]	1368.67	[24, 36]
165-r-c	[**68**, -]	-	[50, **212**]	3120.16	[68, 212]
167-r-e	[**73**, -]	-	[54, **204**]	2112.65	[73, 204]
173-r-c	[**65**, -]	-	[45, **212**]	2004.93	[65, 212]
173-rc-c	[**58**, 172]	2994.35	[37, **168**]	2592.28	[58, 168]
181-r-e	[**82**, -]	-	[55, **216**]	3342.10	[82, 216]
185-c-c	[**32**, -]	-	[14, **96**]	1280.86	[32, 96]
187-rc-e	[**74**, -]	-	[46, **212**]	2849.33	[74, 212]
198-c-c	[**22**, **36**]	158.92	[8, 68]	108.97	[22, 36]
200-r-e	[**77**, -]	-	[48, **252**]	1817.10	[77, 252]
Average	[49.90, -]	-	[34.43, 137.87]	1451.37	[49.90, 120.40]

[c] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

Table 23: Experimental results on the PDSVRP-c. Large instances, 4 trucks.

Instance	$CP2+(47)^c$ [LB, UB]	$CP2+(47)^c$ Sec_{bst}	$CP3+(47)^c$ [LB, UB]	$CP3+(47)^c$ Sec_{bst}	Best bounds
50-r-e	[**46**, **104**]	649.55	[35, 112]	213.25	[46, 104]
53-r-e	[**50**, **96**]	1068.12	[38, 112]	548.64	[50, 96]
66-rc-e	[**41**, **104**]	3493.30	[34, 108]	1019.85	[41, 104]
67-c-c	[**21**, **48**]	25.68	[8, 52]	1297.51	[21, 48]
68-rc-c	[**32**, **52**]	410.57	[29, 88]	296.61	[32, 52]
76-c-c	[**14**, **24**]	44.33	[12, 56]	24.41	[14, 24]
82-c-e	[**18**, **64**]	18.15	[8, **64**]	20.44	[18, 64]
82-rc-c	[**38**, **68**]	2275.00	[31, 124]	194.26	[38, 68]
88-c-e	[28, **76**]	76.88	[**32**, 108]	1177.87	[32, 76]
91-r-c	[**45**, **96**]	3019.26	[32, 156]	248.69	[45, 96]
99-rc-c	[**37**, **68**]	1058.95	[24, 120]	322.23	[37, 68]
101-rc	[**42**, **76**]	3171.49	[30, 144]	2589.25	[42, 76]
103-rc-c	[**39**, **80**]	1490.89	[26, 140]	521.14	[39, 80]
105-rc-e	[**39**, **116**]	261.16	[26, 132]	1691.38	[39, 116]
108-rc-e	[**46**, **124**]	454.82	[28, 152]	3163.13	[46, 124]
114-rc-c	[**35**, **88**]	2564.47	[26, 120]	1369.36	[35, 88]
121-rc-e	[**42**, **104**]	3185.34	[29, 144]	410.13	[42, 104]
126-rc-e	[**50**, **132**]	3362.14	[35, 164]	2600.11	[50, 132]
126-r-c	[**45**, **116**]	1094.09	[28, 140]	729.26	[45, 116]
144-rc-c	[**40**, **128**]	3013.06	[25, 144]	2451.13	[40, 128]
154-c-c	[**18**, **40**]	949.21	[8, 72]	63.77	[18, 40]
165-r-c	[**54**, **192**]	243.15	[40, **192**]	3124.08	[54, 192]
167-r-e	[**58**, **176**]	3277.00	[42, 196]	1489.17	[58, 176]
173-r-c	[**54**, 352]	3435.45	[36, **192**]	3070.29	[54, 192]
173-rc-c	[**46**, **116**]	1650.91	[29, 164]	3368.14	[46, 116]
181-r-e	[**65**, 268]	2937.67	[42, **208**]	3048.86	[65, 208]
185-c-c	[**24**, **48**]	2350.91	[14, 100]	161.08	[24, 48]
187-rc-e	[**58**, 216]	2551.50	[37, **204**]	2097.96	[58, 204]
198-c-c	[**16**, -]	-	[8, **68**]	122.39	[16, 68]
200-r-e	[**60**, 308]	3550.81	[38, **228**]	2613.35	[60, 228]
Average	[40.03, -]	-	[27.67, 133.47]	1334.92	[40.17, 107.87]

[c] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

Table 24: Experimental results on the PDSVRP-c. Large instances, 5 trucks.

Instance	$CP2+(47)^c$ [LB, UB]	$CP2+(47)^c$ Sec_{bst}	$CP3+(47)^c$ [LB, UB]	$CP3+(47)^c$ Sec_{bst}	Best bounds
50-r-e	[**47**, **100**]	227.16	[30, 112]	54.55	[47, 100]
53-r-e	[**50**, **92**]	645.51	[32, 112]	667.24	[50, 92]
66-rc-e	[**35**, **100**]	487.89	[24, 120]	482.57	[35, 100]
67-c-c	[**18**, **52**]	64.31	[8, **52**]	1437.96	[18, 52]
68-rc-c	[**28**, **44**]	1047.89	[23, 80]	1776.83	[28, 44]
76-c-c	[**12**, **24**]	67.04	[**12**, 40]	400.81	[12, 24]
82-c-e	[**15**, **64**]	17.43	[6, **64**]	25.43	[15, 64]
82-rc-c	[**32**, **68**]	592.96	[24, 112]	782.49	[32, 68]
88-c-e	[23, **72**]	218.44	[**32**, 108]	109.66	[32, 72]
91-r-c	[**38**, **88**]	3122.27	[28, 124]	3272.57	[38, 88]
99-rc-c	[**32**, **64**]	597.67	[20, 108]	2532.76	[32, 64]
101-rc	[**36**, 112]	532.65	[26, 144]	505.96	[36, 76]
103-rc-c	[**32**, **80**]	1419.11	[22, 120]	3400.80	[32, 80]
105-rc-e	[**33**, **112**]	1282.90	[21, 124]	410.79	[33, 112]
108-rc-e	[**39**, **120**]	957.98	[24, 136]	1566.66	[39, 120]
114-rc-c	[**30**, **64**]	733.92	[22, 96]	299.50	[30, 64]
121-rc-e	[**34**, 116]	1034.38	[24, 128]	3100.10	[34, 104]
126-rc-e	[**41**, **120**]	2562.32	[29, 148]	2626.65	[41, 120]
126-r-c	[**37**, **116**]	1485.59	[24, 144]	807.31	[37, 116]
144-rc-c	[**34**, **104**]	2325.39	[22, 136]	2332.08	[34, 104]
154-c-c	[**15**, **36**]	1719.34	[6, 68]	669.42	[15, 36]
165-r-c	[**47**, 220]	1614.09	[34, **212**]	3294.70	[47, 212]
167-r-e	[**49**, 204]	1884.33	[34, 204]	1667.48	[49, 196]
173-r-c	[**43**, -]	-	[32, 196]	2657.03	[43, 192]
173-rc-c	[**39**, **116**]	2955.97	[24, 164]	3203.29	[39, 116]
181-r-e	[**54**, **204**]	3349.71	[35, **204**]	2369.20	[54, 204]
185-c-c	[**20**, **48**]	1216.37	[12, 60]	2561.48	[20, 48]
187-rc-e	[**48**, **128**]	2645.47	[32, 192]	2310.65	[48, 128]
198-c-c	[**16**, **36**]	487.28	[8, 68]	118.12	[16, 36]
200-r-e	[**52**, 288]	2545.74	[32, **216**]	2152.81	[52, 216]
Average	[34.30, -]	-	[23.40, 126.40]	1586.56	[34.60, 101.60]

[c] CPU Intel Core i7 12700F - 4x3.6 GHz, 8x4.9 GHz, 20 threads; RAM 32 GB; OR-Tools CP-SAT 9.6; 3600 sec time limit

6.4 Final Remarks

This work introduces several advances for the Parallel Drone Scheduling Traveling Salesman Problem with cooperative drones. We naturally extend the problem by considering multiple vehicles for delivery and introduce a valid inequality that enhances the performance of exact methods. Notably, our research establishes that Constraint Programming approaches can achieve state-of-the-art results for both single- and multiple-vehicle cases.

Furthermore, we underscore the significance of developing effective methodologies that leverage freely available software to tackle these and other combinatorial problems. This emphasis on accessible tools is particularly crucial in the Operations Research context, where the public sharing of code is uncommon.

7 Conclusions

Based on our experience and the results of our contributions, we envision that learning-based methodologies could serve as valuable tools to enhance the resolution of intricate combinatorial optimization problems. While part of our contribution tackles challenges in applying supervised methodologies to such problems, we acknowledge that such methodologies, including Reinforcement Learning ones, have not yet reached a level of maturity to be deemed an effective resolution means. Learning methods currently lag behind traditional state-of-the-art methodologies in terms of performance. Additionally, they pose challenges in terms of ease of application and versatility compared to part of existing methodologies, despite demonstrating superior performance. We believe a common pitfall contributing to sub-optimal performance and complexity in adoption is that learning techniques have been extensively developed for problems significantly different from combinatorial ones. Closing this gap requires further research efforts and new, modified approaches, as exemplified by our contributions in sections 3 and 4.

Furthermore, our work introduces novel and modified scheduling problems that may offer unique opportunities to innovate and enhance methodologies in the context of realistic industrial scenarios. For instance, the workforce constraint limiting the number of simultaneously active machines in Section 5 – a rather practical requirement – breaks the conventional concept of the critical path, rendering well-known and effective neighborhood-based meta-heuristics hardly applicable without cumbersome adaptations (an issue we are currently working on). Additionally, exploring mixed problems involving the scheduling of activities and the planning of other operations may introduce new challenges in standard methodologies, requiring the creation of new hybrid methods that do not increase too much the complexity for broad adoption.

Overall, we believe that our work helps in making a step forward in enriching the repertoire of problem-solving methodologies and also contributes to the broader understanding of adapting computational techniques to the complexity of real-world industrial challenges.

References

[1] Jian Zhang, Guofu Ding, Yisheng Zou, Shengfeng Qin, and Jianlin Fu. Review of job shop scheduling research and its new perspectives under industry 4.0. *Journal of Intelligent Manufacturing*, 30:1809–1830, 2019.

[2] Amit Kumar Gupta and Appa Iyer Sivakumar. Job shop scheduling techniques in semiconductor manufacturing. *The International Journal of Advanced Manufacturing Technology*, 27:1163–1169, 2006.

[3] Michael L Pinedo. *Scheduling: Theory, Algorithms, and Systems*, volume 29. Springer, 2016.

[4] Wen-Yang Ku and J. Christopher Beck. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, 73:165–173, 2016.

[5] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[6] Peter J. M. van Laarhoven, Emile H. L. Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125, 1992.

[7] Eugeniusz Nowicki and Czeslaw Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[8] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms, part ii: hybrid genetic search strategies. *Computers & Industrial Engineering*, 36(2):343–364, 1999.

[9] Kuo-Ling Huang and Ching-Jong Liao. Ant colony optimization combined with taboo search for the job shop scheduling problem. *Computers & Operations Research*, 35(4):1030–1046, 2008.

[10] Libing Wang, Xin Hu, Yin Wang, Sujie Xu, Shijun Ma, Kexin Yang, Zhijun Liu, and Weidong Wang. Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning. *Computer Networks*, 190:107969, 2021.

[11] Reinhard Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, 1989.

[12] Andrea Corsini, Simone Calderara, and Mauro Dell'Amico. Learning the quality of machine permutations in job shop scheduling. *IEEE Access*, 10:99541–99552, 2022.

[13] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1621–1632, 2020.

[14] Zangir Iklassov, Dmitrii Medvedev, Ruben Solozabal, and Martin Takáč. Learning to generalize dispatching rules on the job shop scheduling. *ArXiv*, 2022.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[16] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.

[17] Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management science*, 34(3):391–401, 1988.

[18] Xiao Liu, Fanjin Zhang, Zhenyu Hou, Li Mian, Zhaoyu Wang, Jing Zhang, and Jie Tang. Self-supervised learning: Generative or contrastive. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):857–876, 2023.

[19] Jesper E Van Engelen and Holger H Hoos. A survey on semi-supervised learning. *Machine learning*, 109(2):373–440, 2020.

[20] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pages 4348–4355. International Joint Conferences on Artificial Intelligence Organization, 2021.

[21] C. C. Murray and A. G. Chu. The flying sidekick traveling salesman problem: Optimization of drone-assisted parcel delivery. *Transportation Research Part C: Emerging Technologies*, 54:86–109, 2015.

[22] M. A. Nguyen and M. H. Hà. The parallel drone scheduling traveling salesman problem with collective drones. *Transportation Science*, 4(57):866–888, 2023.

[23] Bahman Naderi, Rubén Ruiz, and Vahid Roshanaei. Mixed-integer programming vs. constraint programming for shop scheduling problems: New results and outlook. *INFORMS Journal on Computing*, 2023.

[24] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[25] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134, 2021.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[27] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *International Conference on Learning Representations (ICLR)*, 2016.

[28] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[29] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[30] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[31] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[34] Razieh Sheikhpour, Mehdi Agha Sarram, Sajjad Gharaghani, and Mohammad Ali Zare Chahooki. A survey on semi-supervised feature selection methods. *Pattern Recognition*, 64:141–158, 2017.

[35] Andrew Goldberg, Xiaojin Zhu, Aarti Singh, Zhiting Xu, and Robert Nowak. Multi-manifold semi-supervised learning. In *Artificial intelligence and statistics*, pages 169–176. PMLR, 2009.

[36] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *International Conference on Learning Representations*, 2018.

[37] David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, and Colin A Raffel. Mixmatch: A holistic approach to semi-supervised learning. *Advances in neural information processing systems*, 32, 2019.

[38] Dong-Hyun Lee et al. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on challenges in representation learning, ICML*, volume 3, page 896, 2013.

[39] Isaac Triguero, Salvador García, and Francisco Herrera. Self-labeled techniques for semi-supervised learning: taxonomy, software and empirical study. *Knowledge and Information systems*, 42:245–284, 2015.

[40] Yaochen Xie, Zhao Xu, Jingtun Zhang, Zhengyang Wang, and Shuiwang Ji. Self-supervised learning of graph neural networks: A unified review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2):2412–2429, 2023.

[41] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[42] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[43] Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. Deep clustering for unsupervised learning of visual features. In *Proceedings of the European conference on computer vision*, pages 132–149, 2018.

[44] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16000–16009, 2022.

[45] Zhenyu Hou, Xiao Liu, Yukuo Cen, Yuxiao Dong, Hongxia Yang, Chunjie Wang, and Jie Tang. Graphmae: Self-supervised masked graph autoencoders. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, page 594–604, 2022.

[46] Haonan Duan, Pashootan Vaezipoor, Max B Paulus, Yangjun Ruan, and Chris Maddison. Augment with care: Contrastive learning for combinatorial problems. In *International Conference on Machine Learning*, pages 5627–5642. PMLR, 2022.

[47] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[48] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

[49] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2018.

[50] Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *ArXiv Preprint*, 2019.

[51] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *ArXiv preprint*, 2016.

[52] André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient active search for combinatorial optimization problems. In *International Conference on Learning Representations*, 2022.

[53] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

[54] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.

[55] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[56] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. *Advances in neural information processing systems*, 28, 2015.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[58] Ruiqi Chen, Wenxin Li, and Hongbing Yang. A deep reinforcement learning framework based on an attention mechanism and disjunctive graph embedding for the job-shop scheduling problem. *IEEE Transactions on Industrial Informatics*, 19(2):1322–1331, 2023.

[59] Egon Balas. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Operations research*, 17(6):941–957, 1969.

[60] Runlong Zhou, Yuandong Tian, Yi Wu, and Simon Shaolei Du. Understanding curriculum learning in policy optimization for online combinatorial optimization. In *OPT 2022: Optimization for Machine Learning (NeurIPS 2022 Workshop)*, 2022.

[61] Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):4555–4576, 2021.

[62] Wiem Mouelhi-Chibani and Henri Pierreval. Training a neural network to select dispatching rules in real time. *Computers & Industrial Engineering*, 58(2):249–256, 2010. Scheduling in Healthcare and Industrial Systems.

[63] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003.

[64] El-Ghazali Talbi. *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009.

[65] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8:239–287, 2009.

[66] Kenneth Sörensen. Metaheuristics—the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[67] Fred Glover and Manuel Laguna. *Tabu Search*, pages 2093–2229. Springer US, Boston, MA, 1998.

[68] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[69] Emile HL Aarts, Peter JM van Laarhoven, Jan Karel Lenstra, and Nico LJ Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing*, 6(2):118–125, 1994.

[70] Pradnya A. Vikhar. Evolutionary algorithms: A critical review and its future prospects. In *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication*, pages 261–265, 2016.

[71] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. volume 80, 2021.

[72] Mauro Dell'Amico and Marco Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operation Research*, 41(1–4):231–252, 1993.

[73] ChaoYong Zhang, PeiGen Li, ZaiLin Guan, and YunQing Rao. A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. *Computers & Operations Research*, 34(11):3229–3242, 2007.

[74] Eugeniusz Nowicki and Czeslaw Smutnicki. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, 8:145–159, 2005.

[75] D.Y. Sha and Cheng-Yu Hsu. A hybrid particle swarm optimization for job shop scheduling problem. *Computers & Industrial Engineering*, 51(4):791–808, 2006.

[76] Helga Ingimundardottir and Thomas Philip Runarsson. Discovering dispatching rules from data using imitation learning: A case study for the job-shop problem. *Journal of Scheduling*, 21(4):413–428, 2018.

[77] James Kotary, Ferdinando Fioretto, and Pascal Van Hentenryck. Fast approximations for job shop scheduling: A lagrangian dual deep learning method. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 7239–7246, 2022.

[78] Chien-Liang Liu, Chuan-Chin Chang, and Chun-Jan Tseng. Actor-critic deep reinforcement learning for solving job shop scheduling problems. *IEEE Access*, 8:71752–71762, 2020.

[79] Junyoung Park, Jaehyeong Chun, Sang Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59:1–18, 2021.

[80] Bao-An Han and Jian-Jun Yang. Research on adaptive job shop scheduling problems based on dueling double dqn. *IEEE Access*, 8:186474–186495, 2020.

[81] El-Ghazali Talbi. Machine learning into metaheuristics: A survey and taxonomy. *ACM Computing Surveys (CSUR)*, 54(6):1–32, 2021.

[82] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[83] Jonas K Falkner, Daniela Thyssens, Ahmad Bdeir, and Lars Schmidt-Thieme. Learning to control local search for combinatorial optimization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 361–376. Springer, 2022.

[84] Simon Thevenin and Nicolas Zufferey. Learning variable neighborhood search for a scheduling problem with time windows and rejections. *Discrete Applied Mathematics*, 261:344–353, 2019. GO X Meeting, Rigi Kaltbad (CH), July 10–14, 2016.

[85] Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. An end-to-end reinforcement learning approach for job-shop scheduling problems based on constraint programming. *Proceedings of the International Conference on Automated Planning and Scheduling*, 33, 2023.

[86] Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *J. Comput. Phys.*, 159(2):139–171, 2000.

[87] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December*, 2014.

[88] Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285, 1993.

[89] Sadegh Mirshekarian and Dušan N. Šormaz. Correlation of job-shop scheduling problem features with scheduling efficiency. *Expert Systems with Applications*, 62:131–147, 2016.

[90] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[91] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[92] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[93] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *INFORMS Journal on Computing*, 3:149–156, 1991.

[94] Achraf Oussidi and Azeddine Elhassouny. Deep generative models: Survey. In *2018 International Conference on Intelligent Systems and Computer Vision*, pages 1–8, 2018.

[95] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? In *International Conference on Learning Representations*, 2022.

[96] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *International Conference on Learning Representations*, 2017.

[97] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134, 2005.

[98] Zichen Zhang, Jun Jin, Martin Jagersand, Jun Luo, and Dale Schuurmans. A simple decentralized cross-entropy method. *Advances in Neural Information Processing Systems*, 35, 2022.

[99] Zdravko I. Botev, Dirk P. Kroese, Reuven Y. Rubinstein, and Pierre L'Ecuyer. The cross-entropy method for optimization. In *Handbook of Statistics*, volume 31. Elsevier, 2013.

[100] Yuki M. Asano, Christian Rupprecht, and Andrea Vedaldi. Self-labelling via simultaneous clustering and representation learning. In *International Conference on Learning Representations*, 2020.

[101] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998.

[102] Jonas K. Falkner, Daniela Thyssens, Ahmad Bdeir, and Lars Schmidt-Thieme. Learning to control local search for combinatorial optimization. In *Machine Learning and Knowledge Discovery in Databases*, pages 361–376, 2023.

[103] Daniel Alejandro Rossit, Fernando Tohmé, and Mariano Frutos. The non-permutation flow-shop scheduling problem: A literature review. *Omega*, 77:143–153, 2018.

[104] Imma Ribas, Rainer Leisten, and Jose M. Framinan. Review and classification of hybrid flow shop scheduling problems from a production system and a solutions procedure perspective. *Computers & Operations Research*, 37(8):1439–1454, 2010. Operations Research and Data Mining in Biological Systems.

[105] Ethel Mokotoff. Parallel machine scheduling problems: A survey. *Asia-Pacific Journal of Operational Research*, 18(2):193, 2001.

[106] Sönke Hartmann and Dirk Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 297(1):1–14, 2022.

[107] Jacek Błażewicz, Klaus H Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. *Handbook on scheduling: from theory to applications*. Springer Science & Business Media, 2007.

[108] Daniel Quadt and Heinrich Kuhn. A taxonomy of flexible flow line scheduling procedures. *European Journal of Operational Research*, 178(3):686 – 698, 2007.

[109] Michael L Pinedo. *Scheduling*, volume 29. Springer, 2012.

[110] Ruben Ruiz and Jose Antonio Vazquez-Rodriguez. The hybrid flow shop scheduling problem. *European Journal of Operational Research*, 205(1):1–18, 2010.

[111] H. Süral, S. Kondakci, and N. Erkip. Scheduling unit-time tasks in renewable resource constrained flowshops. *Zeitschrift für Operations Research*, 36:497–516, 1992.

[112] A. Costa, V. Fernandez-Viagas, and J.M. Framinan. Solving the hybrid flow shop scheduling problem with limited human resource constraint. *Computers & Industrial Engineering*, 146:106545, 2020.

[113] M.K. Marichelvam, M. Geetha, and Omur Tosun. An improved particle swarm optimization algorithm to solve hybrid flowshop scheduling problems with the effect of human factors: a case study. *Computers & Operations Research*, 114:104812, 2020.

[114] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

[115] Dries Bredael and Mario Vanhoucke. Multi-project scheduling: A benchmark analysis of metaheuristic algorithms on various optimisation criteria and due dates. *European Journal of Operational Research*, 2022.

[116] Gaia Nicosia and Andrea Pacifici. Scheduling tasks with comb precedence constraints on dedicated machines. *IFAC Proceedings Volumes*, 46(9):430–435, 2013. 7th IFAC Conference on Manufacturing Modelling, Management, and Control.

[117] Mondher Dhiflaoui, Houssem Eddine Nouri, and Olfa Belkahla Driss. Dual-resource constraints in classical and flexible job shop problems: A state-of-the-art review. *Procedia Computer Science*, 126:1507–1515, 2018. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia.

[118] Hegen Xiong, Shuangyuan Shi, Danni Ren, and Jinjin Hu. A survey of job shop scheduling problem: The types and models. *Computers & Operations Research*, 142:105731, 2022.

[119] A. Alan B. Pritsker, Lawrence J. Watters, and Philip M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, 16(1):93–108, 1969.

[120] Mauro Dell'Amico and Silvano Martello. Optimal scheduling of tasks on identical parallel processors. *Journal on Computing*, 7(2):191–200, 1995.

[121] Reinhard Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, 1989.

[122] Marko Durasevic and Domagoj Jakobovic. A survey of dispatching rules for the dynamic unrelated machines environment. *Expert Systems with Applications*, 113:555–569, 2018.

[123] Shaukat A Brah and Luan Luan Loo. Heuristics for scheduling in a flow shop with multiple processors. *European Journal of Operational Research*, 113(1):113–122, 1999.

[124] Rubén Ruiz and Thomas Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[125] Victor Fernandez-Viagas, Paz Perez-Gonzalez, and Jose M. Framinan. Efficiency of the solution representations for the hybrid flow shop scheduling problem with makespan objective. *Computers & Operations Research*, 109:77–88, 2019.

[126] ZiYan Zhao, MengChu Zhou, and ShiXin Liu. Iterated greedy algorithms for flow-shop scheduling problems: A tutorial. *IEEE Transactions on Automation Science and Engineering*, 19(3):1941–1959, 2022.

[127] Victor Fernandez-Viagas. A speed-up procedure for the hybrid flow shop scheduling problem. *Expert Systems with Applications*, 187:115903, 2022.

[128] Muhammad Nawaz, E Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

[129] Statista. E-commerce. https://www.statista.com/markets/413/e-commerce/, 2022.

[130] Forbes. Drone explosion: $5B investment in 2 years, 129 startups, 170 new craft. https://www.forbes.com/sites/johnkoetsier/2022/02/07/drone-innovation-check-up-5b-investment-129-companies-170-craft/, 2022.

[131] R. Wolleswinkel, V. Lukic, W. Jap, R. Chan, J. Govers, and S. Banerjee. *An onslaught of new rivals in parcel and express*, volume Travel, Transport and Logistics. Boston Consulting Group, 2018.

[132] M. Dell'Amico, R. Montemanni, and S. Novellani. Algorithms based on branch and bound for the flying sidekick traveling salesman problem. *Omega*, 104:102493, 2021.

[133] M. Dell'Amico, R. Montemanni, and S. Novellani. Exact models for the flying sidekick traveling salesman problem. *Omega*, 29(3):1360–1393, 2022.

[134] R. G. Mbiadou Saleu, L. Deroussi, D. Feillet, N. Grangeon, and A. Quilliot. An iterative two-step heuristic for the parallel drone scheduling traveling salesman problem. *Networks*, 72(4):459–474, 2018.

[135] M. Dell'Amico, R. Montemanni, and S. Novellani. Matheuristic algorithms for the parallel drone scheduling traveling salesman problem. *Annals of Operations Research*, 289:211–226, 2020.

[136] Q. T. Dinh, D. D. Do, and M. H. Hà. Ants can solve the parallel drone scheduling traveling salesman problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 14–21, 2021.

[137] D. Lei and X. Chen. An improved variable neighborhood search for parallel drone scheduling traveling salesman problem. *Applied Soft Computing*, 127:109416, 2022.

[138] R. Montemanni and M. Dell'Amico. Solving the parallel drone scheduling traveling salesman problem via constraint programming. *Algorithms*, 16(1):40, 2023.

[139] M. A. Nguyen, H. L. Luong, M. H. Hà, and H. B. Ban. An efficient branch-and-cut algorithm for the parallel drone scheduling traveling salesman problem. *4OR*, 2022.

[140] A. Otto, N. Agatz, J. Campbell, B. Golden, and E. Pesch. Optimization approaches for civil applications of unmanned aerial vehicles (uavs) or aerial drones: A survey. *Networks*, 72(4):411–458, 2018.

[141] J. Pasha, Z. Elmi, S. Purkayastha, A. M. Fathollahi-Fard, Y.-E. Ge, Y.-Y. Lau, and M. A. Dulebenets. The drone scheduling problem: A systematic state-of-the-art review. *IEEE Transactions on Intelligent Transportation Systems*, 23(9):14224–14247, 2022.

[142] R. G. Mbiadou Saleu, D. Deroussi, L. qnd Feillet, N. Grangeon, and A. Quilliot. The parallel drone scheduling problem with multiple drones and vehicles. *European Journal of Operational Research*, 300:571–589, 2022.

[143] R. Raj, D. Lee, S. Lee, J. Walteros, and C. Murray. A branch-and-price approach for the parallel drone scheduling vehicle routing problem. *SSRN Electronic Journal*, pages 1–47, 2021.

[144] M. A. Nguyen, G. T.-H. Dang, M. H. Hà, and M.-T. Pham. The min-cost parallel drone scheduling vehicle routing problem. *European Journal of Operational Research*, 299:910–930, 2022.

[145] R. Montemanni and M. Dell'Amico. Constraint programming models for the parallel drone scheduling vehicle routing problem. *EURO Journal on Computational Optimization*, 11:100078, 2023.

[146] R. Raj and C. Murray. The multiple flying sidekicks traveling salesman problem with variable drone speeds. *Transportation Research Part C*, 120:102813, 2020.

[147] Z. Liu, R. Sengupta, and A. Kurzhanskiy. A power consumption model for multi-rotor small unmanned aircraft systems. In *Proceedings of the IEEE International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 310–315, 2017.

[148] J. Zhang, J. F. Campbell, D. C. II Sweeney, and Hupman A. C. Energy consumption models for delivery drones: A comparison and assessment. *Transportation Research Part D*, 90, 2021.

[149] N. M. Paczan, M. J. Elzinga, R. Hsieh, and L. K. Nguyen. Collective unmanned aerial vehicle configurations. *Patent US 11,480,958 B2*, 2022.

[150] L. Perron and V. Furnon. Google OR-Tools, 2023. https://developers.google.com/optimization/ [Accessed: 2023-03-03].

[151] P. J. Stuckey. Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving. In *Proceedings of the International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR)*, pages 5–9, 2010.

[152] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67, 1997.

[153] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[154] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

[155] IBM ILOG. User's manual for CPLEX. https://www.cplex.com/, 2023.

Acronyms

CO

Combinatorial Optimization.

CP

Constraint Programming.

FFSP

Flexible Flow Shop Scheduling.

FNN

Feedforward Neural Network.

FSP

Flow Shop Scheduling.

GNN

Graph Neural Network.

Js-IG

Job Sequencing - Iterated Greedy.

JSP

Job Shop Scheduling.

MILP

Mixed Integer Linear Programming.

ML

Machine Learning.

oTS

Oracle-based Tabu Search.

PDR

Priority Dispatching Rule.

PDR-IG

Priority Dispatching Rule - Iterated Greedy.

PDSTSP

Parallel Drone Scheduling Traveling Salesman Problem.

PDSTSP-c

Parallel Drone Scheduling Traveling Salesman Problem with Collective Drones.

PDSVRP-c

Parallel Drone Scheduling Vehicle Routing Problem with Collective Drones.

PG

Percentage Gap.

PN

Pointer Network.

RCPSP

Resource-Constrained Project Scheduling Problem.

RL

Reinforcement Learning.

RNN

Recurrent Neural Network.

rTRCFFSP

rooted in-Tree Resource Constrained Flexible Flow Shop Scheduling.

SPN

Self-Labeling Pointer Network.

TS

Tabu Search.

TSP

Traveling Salesman Problem.

WTA

Within Tolerance Accuracy.