EURO Journal on Computational Optimization 13 (2025) 100118

Contents lists available at ScienceDirect

EURO Journal on Computational Optimization

journal homepage: www.elsevier.com/locate/ejco

Investigating the Monte-Carlo Tree Search approach for the job shop scheduling problem

Laurie Boveroux*, Damien Ernst, Quentin Louveaux

University of Liège, Allée de la Découverte 10, 4000, Liège, Belgium

HIGHLIGHTS

- New benchmark of large-scale complex industrial JSSP instances with uneven workloads.
- Different MDP formulations to represent the JSSP.
- Monte-Carlo Tree Search demonstrates its effectiveness in solving largescale JSSP.

ARTICLE INFO

Keywords:
Job shop scheduling
Monte-Carlo Tree Search

ABSTRACT

The Job Shop Scheduling Problem (JSSP) is a well-known optimization problem in manufacturing, where the goal is to determine the optimal sequence of jobs across different machines to minimize a given objective. In this work, we focus on minimizing the weighted sum of job completion times. We explore the potential of Monte Carlo Tree Search (MCTS), a heuristic-based reinforcement learning technique, to solve large-scale JSSPs, especially those with recirculation. We propose several Markov Decision Process (MDP) formulations to model the JSSP for the MCTS algorithm. In addition, we introduce a new synthetic benchmark derived from real manufacturing data, which captures the computational burden of large, non-rectangular instances often encountered in practice. Our experimental results show that MCTS effectively produces good-quality solutions for large-scale JSSP instances, outperforming our constraint programming approach.

1. Introduction

The Job Shop Scheduling Problem (JSSP) is a complex challenge faced by manufacturers. The JSSP involves determining the optimal sequence of jobs on different machines to ensure that production processes are carried out efficiently. This problem is important because it directly impacts a company's productivity, operating costs and ability to meet delivery schedules. For example, delays in the production schedule can cause bottlenecks, higher inventory costs, and missed deadlines. These issues can lead to unhappy customers and financial penalties. Therefore, optimizing job scheduling is a practical need and a decisive strategy for success.

Many mathematical programming-based approaches exist to solve the JSSP, such as mixed-integer linear programming and constraint programming [1]. These methods are exact as they find optimal solutions by exhaustively exploring the search space. However, they have limitations in practice. The JSSP is known to be NP-hard [2]. As the number of jobs and machines increases, the complexity

* Corresponding author.
Email addresses: laurie.boveroux@uliege.be (L. Boveroux), dernst@uliege.be (D. Ernst), q.louveaux@uliege.be (Q. Louveaux).

https://doi.org/10.1016/j.ejco.2025.100118

Available online 9 October 2025

of the problem grows exponentially. In practice, scheduling problems are often large-scale, dynamic and unbalanced. In such scenarios, some machines may be heavily loaded while others are idle and processing times of the tasks can vary widely from a few units of time to several hundred units. In these large-scale environments, exact methods become impractical.

To address these limitations, approximate solutions have been developed. Commonly used heuristics are Priority Dispatching Rules (PDRs) [1]. PDRs are simple heuristics that select the next operation to be scheduled based on a specific criterion. These rules are easy to implement and computationally efficient, but they often provide low-quality solutions. Another well-known heuristic approach is the Shifting Bottleneck Heuristic [3]. This procedure solves the problem by iteratively identifying the machine that creates a bottleneck and solving the schedule optimally on that unique machine using the one-machine scheduling method by Carlier [4].

Different meta-heuristic approaches such as simulated annealing, tabu search, genetic algorithms and a variety of nature-inspired meta-heuristics have been proposed over the years [5]. Simulated annealing is simple to implement and effective at escaping local optima, but it can converge slowly and is sensitive to parameter tuning. Tabu search provides strong local search performance and avoids cycling via memory structures, yet it can be computationally intensive and requires careful parameter tuning. Genetic algorithms are global optimisers able to explore diverse solution regions, though they may experience premature convergence and slow fine-tuning. Nature-inspired metaheuristics are flexible, scalable and can handle complex constraints, but often rely on precise parameter control and may incur high computational costs before convergence. A recent comprehensive review [5] compared these approaches in terms of their main concept, advantages, disadvantages and performance across multiple JSSP variants, underscoring that no single algorithm dominates in all scenarios. Performance depends on problem characteristics, tuning and hybrid strategies. Additionally, the authors note that the lack of standardised benchmark datasets and performance evaluation metrics remains a major obstacle to fair comparison and validation of proposed methods.

Beyond metaheuristic approaches, recent research has explored learning-based methods for the JSSP, where scheduling policies are derived automatically from data or simulated experience. One such method is deep reinforcement learning, which can represent job-machine interactions and produce scheduling decisions. For instance, Zhang et al. [6] use a graph neural network with an actor-critic algorithm to learn dispatching rules.

One promising approach is the Monte Carlo Tree Search (MCTS) algorithm. MCTS is a heuristic search algorithm that combines tree search with random sampling to find solutions in large search spaces. Initially applied to the game domain, this reinforcement learning (RL) algorithm has shown its effectiveness in finding good strategies for complex games such as chess and Go [7]. The JSSP shares similarities with these games, as both involve making sequential decisions and optimizing outcomes based on a series of actions. More specifically, we note that MCTS algorithms have been applied successfully to solve Markov Decision Processes (MDPs) in domains such as deterministic board games (Go, Chess, Shogi), puzzles (SameGame, Sudoku) and real-time video games (Pac-Man, Starcraft) [8], which suggests their applicability to JSSP as they can be effectively framed as an MDP. In this formulation, a scheduling problem can be represented in the same way as these domains: states encode the current status of jobs and machines, actions correspond to scheduling decisions, transitions describe how the system evolves after each decision and rewards capture the optimization objective. Different MDP formulations can be employed to represent the scheduling problems.

Several studies have applied MCTS to job-shop and flexible job-shop scheduling by building a schedule directly in a step-by-step fashion. Examples include the works of Runarsson et al. [9], Chou et al. [10], Wu et al. [11], Lubosch et al. [12] and Saqlain et al. [13]. In these methods, each scheduling decision adds one or several tasks to the schedule in a specific position on a machine, and the process repeats until all tasks are placed. The search algorithm evaluates different sequences of such decisions to find high-quality schedules. Alongside these direct construction approaches, there also exist hybrid methods that combine MCTS with other optimization techniques. For instance, Loth et al. [14] integrate MCTS with constraint programming, where each action assigns a value to a decision variable in a constraint model. The CP solver then propagates constraints.

While a wide variety of approaches have been applied to the JSSP, their evaluation is typically conducted on traditional benchmark instances such as those proposed by Taillard et al. [15] and Adams et al. [3]. These benchmarks do not capture the computational burden of real-world scenarios, where machine loads can vary significantly, with some machines heavily loaded while others are idle. These benchmarks are less suitable for evaluating algorithm performance in large-scale industrial scheduling problems. To address this gap, we introduce a new synthetic benchmark derived from real-world manufacturing data that captures the complexity typical of large-scale industrial environments. In this work, we investigate the potential of MCTS in solving large-scale JSSP. The main contributions of this work are:

1. We investigate and evaluate different ways to model JSSP as an MDP for the MCTS algorithm.
2. We deliver a new benchmark created from anonymised real-world data.[1]

The rest of the paper is organized as follows. In Section 2, we formally define the JSSP. Section 3 describes our approach, including how we model JSSP as an MDP for MCTS and the constraint programming model used for comparison. In Section 4, we explain the process of generating a new benchmark from anonymised real-world data. Section 5 presents the experimental setup and results. Finally, Section 6 concludes the paper and outlines directions for future research.

[1] https://github.com/LaurieBvrx/large-scale-complex-JSSP-benchmark.git.

2. Problem statement

In this work, we focus on the Job Shop Scheduling Problem (JSSP). In a general JSSP, we are given a set $\mathcal{J}$ of n jobs $J_1, J_2, \ldots, J_n$ and a set $\mathcal{M}$ of m machines. Each job $J_i \in \mathcal{J}$ has an operation set $\mathcal{O}_i$ which contains n_i operations O_{ij} that must be processed in a specific order (i.e., with precedence constraints). Each operation O_{ij} of job J_i requires a processing time p_{ij} on a specific machine $M_{(ij)}$. A job can have several operations that must be processed on the same machine (i.e., recirculation). Each machine can process at most one operation at a time with no preemption.

To solve the JSSP, we must find a schedule that determines the order in which the operations are processed to minimize a specific objective function. One of the most widely studied objectives in the literature is the makespan, defined as the maximum completion time of all operations in the schedule. This criterion is particularly relevant when the goal is to minimize the total production time. In our context, we are interested in a different objective that reflects a certain industrial priority: the weighted sum of job completion times. This measure accounts for the time each job spends in the system and its relative importance, aligning with situations where companies aim to maximize billings over time and reduce work-in-progress inventory. It can be formulated as follows:

$$\text{minimize} \quad \sum_{j \in \mathcal{J}} w_j \, C_j$$

where w_j is the weight of the job j and C_j its completion time.

To describe a scheduling problem accurately, the standard notation in literature is the triplet $\alpha|\beta|\gamma$ where α represents the machine environment, β the processing characteristics and the constraints and γ the objective. The problem we consider can be characterized by the triplet

$$J_m|prec, rcrc|w_j C_j$$

This triplet refers to a job shop environment with m machines (J_m). There are precedence constraints ($prec$), meaning that there are certain jobs or operations that must be completed before others can begin. The other processing characteristic is the recirculation ($rcrc$), which implies that two or more operations of a job can be processed on the same machine. In contrast to a classic job shop, where each job has exactly one operation on each machine, recirculation allows jobs to visit a machine more than once. Finally, the objective is the minimization of the total weighted completion times.

3. Approach

In this section, we first recall the MDP concepts used by MCTS, then outline the standard MCTS algorithm. Next, we cast the JSSP as an MDP and propose four action-space designs. We also detail the PDRs we use both as standalone baselines and as policies inside the action spaces and we include a CP approach for a stronger comparison. Finally, we illustrate one complete MCTS iteration on a toy instance.

3.1. Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm used to make decisions in environments that can be modelled as a *Markov Decision Process* (MDP) [8]. An MDP is defined by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, transition probabilities $p(s'|s,a)$ describing how the environment evolves after taking action a in state s, and a reward function $r(s,a)$ that quantifies the immediate benefit of an action. MCTS provides a way to approximate this optimal decision-making policy by incrementally building a search tree through repeated simulations.

Each node in the tree represents a state $s \in \mathcal{S}$, and each edge represents an action $a \in \mathcal{A}$ leading to a successor state s'. The algorithm balances exploration (trying new actions) and exploitation (choosing the best-known action) through simulations and updates. The procedure consists of four key steps, repeated iteratively N_{repeat} times. These are schematically represented in Fig. 1 [8].

1. **Selection:** Go through the tree from the root until reaching a node that is not fully expanded using a tree policy, typically the Upper Confidence Bound (UCT) formula. At each step, the child node i with the highest UCT score is selected. The score is given by:

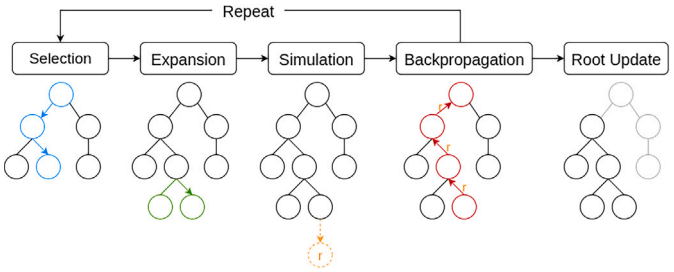

Fig. 1. The four fundamental steps of the MCTS algorithm and the additional step.

$$\text{UCT} = \frac{W_i}{N_i} + C\sqrt{\frac{\ln N}{N_i}}$$

where W_i is the total reward of child node i, N_i is the number of times node i has been visited, N is the total number of visits to the parent node, and C is a constant. This formula encourages selecting nodes that have either high average rewards (exploitation) or low visit counts (exploration).

2. **Expansion**: If the selected node is not terminal, expand its children corresponding to possible actions. One child is chosen randomly.
3. **Simulation**: From the selected node, perform a rollout using a default policy (typically random) until a terminal state is reached. The sequence of states and actions visited during the simulation is not added to the search tree; only the terminal reward is recorded. Repeat this procedure N_{sim} times and use the best obtained reward to evaluate the node.
4. **Backpropagation**: Update the statistics (e.g., visit counts and rewards) of all nodes on the path from the simulated node back to the root.

In addition to the four standard MCTS steps, a fifth optional step called Root Update can be applied to enhance the search efficiency over time. After several iterations of the first four steps, if the algorithm has thoroughly explored the tree up to a certain depth, it may choose to:

- Select the most promising child node at the current root (e.g., based on the average reward),
- Discard the rest of the tree, and
- Continue the search from that child as the new root.

This allows the search tree to focus the computational effort on the most promising subtree. The detailed steps of the MCTS algorithm are outlined in Algorithm 1, which provides a pseudocode implementation of the process.

3.2. Priority dispatching rules

Going back to the scheduling problem, we now present a class of heuristics: the *Priority Dispatching Rules* (PDRs). They are very popular in the framework of large-scale instances [16] and we will use them in the policies of the MCTS. PDRs are simple decision-making rules that select the next operation or job to be scheduled based on specific criteria. They are widely used in practice because they are easy to implement and computationally efficient, particularly for large-scale problems. Depending on the level of granularity, PDRs can operate either at the job level or at the operation level. At the job level, a PDR prioritizes entire jobs according to certain characteristics (e.g., number of operations). At the operation level, it prioritizes individual operations either based on their own attributes (e.g., processing time) or because they are the first available operation of a job selected by a job-level rule. In all cases, PDRs are applied in a constructive manner: starting from an empty schedule, they iteratively extend a partial schedule by selecting the next job or operation according to the chosen rule, until a complete schedule is obtained.

The list of the job-level PDRs are listed in the following:

- The FIFO rule (first in first out) processes jobs in the order they are given in the instance.
  If J_i and J_j are two jobs, then under FIFO:

  If $i < j$, then $J_i$is processed before J_j.

- The Least Work First (LWF) rule selects the job with the shortest total processing time. The processing time of a job is the sum of the processing times of all its operations. If J_i and J_j are two jobs and P_i represents the total processing time for job J_i, then under LWF:

  If $P_i < P_j$, then $J_i$is scheduled before J_j.

- The Most Work First (MWF) rule selects the job with the longest total processing time.
- The Shortest Job First (SJF) rule prioritizes jobs with the least number of operations.
  If J_i and J_j are two jobs and n_i represents the number of operations of job J_i, then under SJF:

  If $n_i < n_j$, then $J_i$is scheduled before J_j.

- The Largest Job First (LJF) rule selects the job with the largest number of operations.

The list of the operation-level PDRs are listed in the following:

- The FIFO rule (first in first out) processes operations in the order they are given in the instance.
  If O_i and O_j are two operations, then under FIFO:

  If $i < j$, then $O_i$is processed before O_j.

Algorithm 1 Monte Carlo Tree Search.

function MCTS($N_{rootUpdate}$, N_{repeat}, N_{sim})
  $v_0 \leftarrow$ initialize root node
  for $i = 1$ to $N_{rootUpdate}$ **do**
    for $j = 1$ to N_{repeat} **do**
      $v \leftarrow$ SELECT(v_0)
      $v' \leftarrow$ EXPAND(v)
      for $k = 1$ to N_{sim} **do**
        $r \leftarrow$ SIMULATE(v')
        if $r > r^*$ **then**
          $r^* \leftarrow r$
        end if
      end for
      BACKPROPAGATE($v', r*$)
    end for
    $v_0 \leftarrow \arg\max_{v \in \text{Children}(v_0)} W_v / N_v$
  end for
end function

function SELECT(v)
  while $Children(v) \neq$ null **do**
    $v \leftarrow arg \max_{u \in \text{Children}(v)} \text{UCT}(u)$
  end while
  return v
end function

function EXPAND(v)
  for $a \in Actions(v)$ **do**
    Generate successor state $s' = \text{Apply}(v.state, a)$
    Create new node v' with $v'.state = s'$
    Add v' as a child of v
  end for
  return $Random(Children(v))$
end function

function SIMULATE(v)
  $s \leftarrow v.state$
  while s is not terminal **do**
    Choose action a according to default policy (e.g., random)
    $s \leftarrow \text{Apply}(s, a)$
  end while
  return Reward(s)
end function

function BACKPROPAGATE(v, r)
  while $v \neq$ null **do**
    $N_v \leftarrow N_v + 1$
    $W_v \leftarrow W_v + r$
    $v \leftarrow$ parent of v
  end while
end function

- The Least Work Remaining (LWR) rule prioritizes the first available operation of the job with the least total processing time remaining across all jobs. If J_i and J_j are two jobs, O_i and O_j are the first available operations of jobs J_i and J_j and $P_{i,t}$ represents the total processing time remaining for job J_i at step t, then under LWR:

  If $P_{i,t} < P_{j,t}$, then $O_i$is scheduled before O_j.

- The Most Work Remaining (MWR) rule is similar to the LWR rule, but instead of prioritizing the least total processing time, it prioritises the first available operation of the job with the most total processing time remaining.

- The Least Operations Remaining (LOR) rule prioritizes the first available operation of the job with the least number of operations remaining across all jobs.
  If J_i and J_j are two jobs, O_i and O_j are the first available operations of jobs J_i and J_j respectively and $n_{i,t}$ represents the number of operations that still have to be scheduled for job J_i at step t, then under LOR:

  If $n_{i,t} < n_{j,t}$, then $O_i$is scheduled before O_j.

- The Most Operations Remaining (MOR) rule is similar to the LOR rule, but instead of prioritizing the least number of operations, it prioritizes the first available operation of the job with the most operations remaining.
- The Shortest Processing Time (SPT) rule selects the operation with the shortest processing time. If O_i and O_j are two operations and p_i and p_j are their processing times, then under SPT:

  If $p_i < p_j$, then $O_i$is processed before O_j.

- The Longest Processing Time (LPT) rule selects the operation with the longest processing time.

3.3. Modelling JSSP as an MDP

This article explores different environments for the MCTS applied to the JSSP. Each environment is defined by its state space, action space and reward function. These three components can be defined in multiple ways, each influencing the performance and behaviour of the MCTS algorithm differently. We explore a few of the possible combinations of these components in the following.

3.3.1. State space

In reinforcement learning, the state s_t is a representation of the situation of the agent at the decision step t. In the context of JSSP, the state s_t corresponds to the partial schedule after t decision steps. Within MCTS, each state corresponds to a node in the search tree, where the root represents an empty schedule and each path to a leaf represents a complete sequence of scheduling decisions. This partial schedule can be seen as a Gantt chart showing which operations have been assigned to machines and their respective start times. The state thus encapsulates the current progress of the schedule construction, providing all necessary information for selecting the next operation or job to schedule.

3.3.2. Action space

An action space A is a set of possible actions that the agent can take in a given state. In the context of the JSSP, an action determines which operation(s) to schedule next and where to place them on the machines.

Fig. 2 illustrates a path through a toy instance with 3 jobs. At the initial decision step $t = 0$, no operation is scheduled. At each subsequent decision step, an operation is selected (highlighted in red) and scheduled at the earliest feasible time on its corresponding machine. The figure shows snapshots of this scheduling process at decision steps $t = 0, 1, 2, 6, 7$ and a terminal state $t = 10$. This example demonstrates one type of action space, where at each step a single operation is chosen and inserted at the earliest possible start time.

The scheduling decisions in action spaces of this paper are guided by Priority Dispatching Rules (PDRs), which are simple heuristics that prioritize operations or jobs based on specific criteria. Each of the PDRs used in this work is described in detail in Section 3.2.

We propose four types of action spaces, each corresponding to a different scheduling strategy. The first two types of action spaces (Single Operation Selection and Whole Job Selection) follow established approaches in the scheduling and MCTS literature. Whereas the last two (Single Operation Selection with Percentage-Based Insertion and Whole Job Selection with Percentage-Based Insertion) are new formulations that extend the standard earliest-start insertion strategy. Let:

- $\bar{\mathcal{O}}$ = set of unscheduled operations.
- $\bar{\mathcal{J}}$ = set of unscheduled jobs.
- $\mathcal{R}$ = set of dispatching priority rules (PDRs).
- $r(X)$ = element of set X selected by applying PDR $r \in \mathcal{R}$.
- $p \in [0, 1]$ = percentage-based insertion parameter.
- `insertEarliest` = insert chosen operation o at the earliest feasible start time.
- `insertWithGap`(p) = insert chosen operation o into first feasible gap of length at least $p \cdot \text{duration}(o)$.

1. Single operation selection via PDR

   At each decision step, select one operation from the unscheduled set $\bar{\mathcal{O}}$ using a priority dispatching rule (PDR), then insert it at the earliest feasible time:

   $$A_1 = \left\{ a \mid \exists r \in \mathcal{R},\ a = \left(r(\bar{\mathcal{O}}), \texttt{insertEarliest}\right) \right\}.$$

   This corresponds exactly to the action space illustrated by the toy example in Fig. 2. Each red-highlighted operation selected at decision steps $t = 0, 1, 2, \ldots$ is chosen by a PDR and scheduled at the earliest possible time. This strategy schedules one operation per action.

   For example, if $\mathcal{R} = \{FIFO, LWR, MWR\}$, then at a given decision step, the agent might either:

   - choose the operation whose job entered earliest (FIFO),
   - choose the operation belonging to the job with the least total work remaining (LWR), or
   - choose the operation from the job with the most work remaining (MWR).

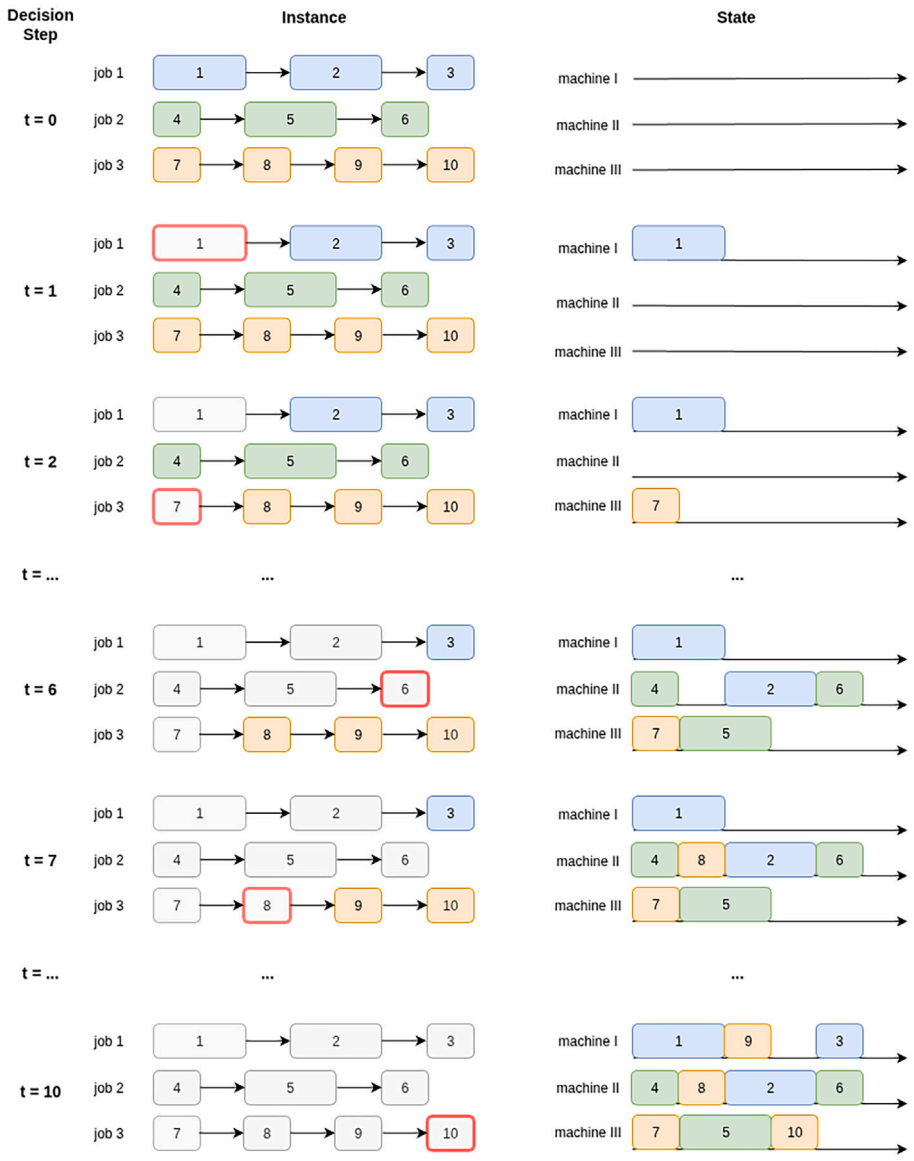

Fig. 2. Scheduling example for three jobs, showing operations (red) placed at earliest start times for $t = 0, 1, 2, 6, 7$ and final state $t = 10$. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Once selected, the agent inserts the operation into the schedule at the earliest feasible start time on its machine, respecting both machine availability and job precedence constraints. Runarsson et al. [9] also follow this single-operation scheduling pattern, where an action consists of selecting a job from the available list and scheduling its next operation. In their pilot and MCTS variants, the job is chosen using a dispatching rule (MWKR, SPT, LOPN). Wu et al. [11] adopt a similar operation-level action definition in the context of a multi-objective flexible job shop scheduling problem. At each decision step, an unscheduled operation is paired with one of its feasible machines, and the pair is scheduled according to a selection policy that balances multiple objectives. Chou et al. [10] also define an action as scheduling a single operation, but without restricting the choice to a predefined set of PDRs. Instead, the MCTS algorithm explores all currently available operations, each paired with one of its N_{best} eligible machines and evaluates them using the UCT formula. Because this action space is much larger than in our A_1 formulation, they apply techniques such as limiting the number of considered machines, pruning subtrees corresponding to dominated partial schedules, and using transposition tables to keep the search tractable. Similarly, Lubosch et al. [12] use MCTS to select individual operations without fixed PDRs, but in their case the exploration–exploitation parameter of the selection function is tuned automatically via Gradient Boosted Decision Trees, so that machine learning influences how actions are selected rather than directly proposing them.

2. Whole job selection via PDR

   Alternatively, an action may select an entire job $j \in \bar{\mathcal{J}}$ using a PDR and schedule all its operations sequentially at the earliest feasible times:

   $$A_2 = \left\{ a \mid \exists r \in \mathcal{R},\ a = \left(r(\bar{\mathcal{J}}), \texttt{insertEarliest}\right) \right\}.$$

   This approach reduces the depth of the search tree by bundling multiple operations into a single action and can make exploration more efficient.

   For example, if $\mathcal{R} = \{FIFO, LWF, MWF\}$, then at a given decision step, the agent might either:

   - choose the job that arrived first (FIFO),
   - choose the job with the least total processing time (LWF), or
   - choose the job with the most total processing time (MWF).

   After selecting the job, all of its operations are inserted sequentially at their earliest feasible start times on their respective machines.

   This type of action space corresponds closely to the one in Saqlain et al.'s MCTS-FJS algorithm for flexible job-shop scheduling [13]. In their work, the MCTS agent selects a scheduling rule (FIFO, SJF or LJF), which chooses the next job to schedule. All operations of that job are then inserted sequentially at their earliest feasible start times on the respective machines, respecting precedence and machine constraints. Our formulation generalizes this approach by allowing an arbitrary set $\mathcal{R}$ of dispatching rules.

3. Single operation selection with percentage-based insertion

   This is the first of our new action space designs. Instead of always placing an operation at its earliest feasible start time, the agent may insert it into the first available machine gap of length at least a fraction p of its processing time.

   $$A_3 = \left\{ a \mid \exists r \in \mathcal{R},\ p \in [0,1],\ a = \left(r(\bar{\mathcal{O}}), \texttt{insertWithGap}(p)\right) \right\}.$$

   After determining where to insert the operation in the schedule, the completion times of this operation and all subsequent ones on the machine must be recalculated by resolving the earliest feasible start and finish times, taking into account precedence and machine constraints. This is necessary because inserting an operation into a gap that is too small may cause a shift in the timing of subsequent operations. The computation of updated completion times can be done in $\mathcal{O}(n \log n)$, where n is the number of operations.

   For example, if $\mathcal{R} = SPT$ and $p \in \{0.6, 0.8, 1.0\}$, the agent selects the operation with the shortest processing time and might insert it into either

   - the first gap at least 60 % of its duration ($p = 0.6$),
   - the first gap at least 80 % of its duration ($p = 0.8$), or
   - only the first gap that can fit its duration ($p = 1.0$).

4. Whole job selection with percentage-based insertion

   This action space extends the previous idea to whole-job scheduling. After selecting a job using a PDR, all of its operations are inserted sequentially using the same percentage-based gap rule.

   $$A_4 = \left\{ a \mid \exists r \in \mathcal{R},\ p \in [0,1],\ a = \left(r(\bar{\mathcal{J}}), \texttt{insertWithGap}(p)\right) \right\}.$$

   This combines the advantages of reduced tree depth from whole-job scheduling and flexible insertion via gaps.

   For example, if $\mathcal{R} = \{LWF\}$ and $p \in \{0.6, 0.8, 1.0\}$, the agent selects the job with the shortest total processing time and schedules all of its operations in sequence. Each operation might be inserted into either:

   - the first gap at least 60 % of its duration ($p = 0.6$),
   - the first gap at least 80 % of its duration ($p = 0.8$), or
   - the first gap that can fit its entire duration ($p = 1.0$).

   Once all operations of the selected job have been inserted, the completion times for these operations and all affected subsequent operations must be recomputed by resolving the earliest feasible start and finish times, considering precedence and machine constraints. Unlike single-operation insertion, the recomputation is performed only once after scheduling the entire job. This recomputation has the same $\mathcal{O}(n \log n)$ complexity as above but is performed only once after scheduling the entire job.

Reward

The reward function is related to the objective function of the problem and is designed to reflect the quality of the solution. The most intuitive way to define the reward function in the context of the JSSP would be to set the reward to 0 unless a terminal state has been reached. A terminal state is a state reached when the problem is solved (i.e., all jobs are scheduled) or it is impossible to continue (for example, due to infeasibility, such as exceeding resource limits or invalid machine assignments). In such cases, the reward is defined differently:

- If the terminal state represents a successfully completed schedule, the reward is the negative weighted sum of the completion times of all jobs, aligning with the objective to minimize the total weighted completion time.
- If the terminal state represents an infeasible solution, the reward is $-\infty$.

Table 1
Cumulative rewards (W_i) and visit counts (N_i) of the nodes in the toy MCTS example before and after applying one iteration.

Node id	A	B	C	D	E	F	G	H	I	J	K
W_i (before)	3.2	1.7	1.5	0.3	0.8	0.3	0.7	0	0.3	∄	∄
N_i (before)	7	4	3	1	2	1	1	0	1	∄	∄
W_i' (after)	3.7	1.7	2.0	0.3	0.8	0.3	1.2	0	0.3	0	0.5
N_i' (after)	8	4	4	1	2	1	2	0	1	0	1

The objective function is a minimization problem, whereas MCTS is formulated as a maximization framework. To make smaller completion times correspond to larger rewards and to comply with the UCT assumption of rewards in the range $[0, 1]$, we apply the following normalization:

$$\hat{r} = \frac{r_{\max} - r}{r_{\max} - r_{\min}}$$

where $r_{\min}$ and $r_{\max}$ are constants providing simple bounds on the reward. $r_{\min}$ corresponds to the total processing time of all operations, i.e., the idealized case where all jobs are processed in parallel without precedence or machine constraints. This is an easily computed lower bound. $r_{\max}$ corresponds to the sum of the completion times when jobs are processed sequentially. In this case, the completion time of job i equals the completion time of job $i-1$ plus the total processing time of all operations of job i. This gives an easily computed upper bound.

3.4. MCTS applied to JSSP

Fig. 4 illustrates one complete iteration of the MCTS algorithm applied to a toy example, assuming that a partial search tree has already been constructed. We first illustrate the partial search tree in Fig. 3 and the statistics of the nodes are given in Table 1.

The action space is defined as

$$A = \left\{ a \mid \exists r \in \{\text{FIFO}, \text{SPT}\},\ a = \left(r(\bar{\mathcal{O}}), \texttt{insertEarliest}\right) \right\}.$$

At the root node n_A, no action has been taken yet and no operations are scheduled. The corresponding Gantt chart is empty. The node n_A has two children, n_B and n_C:

- Node n_B corresponds to applying the FIFO rule. Among the available operations (1, 4 and 7), FIFO selects operation 1. This results in the state s_{AB}.
- Node n_C corresponds to applying the SPT rule. Among the same available operations, SPT selects operation 4. This operation is then scheduled at the earliest possible time on machine II, yielding the state s_{AC}.

Similarly, by applying the action (SPT, `insertEarliest`) to node n_B (i.e., to state s_{AB}), we obtain the state s_{ABE}.

Selection phase. We begin at the root node n_A:

i. We first apply the UCT formula to the children of n_A based on statistics obtained in previous iterations and reported in Table 1.

$$UCT(n_B) = \frac{1.7}{4} + \sqrt{2}\sqrt{\frac{\ln(7)}{4}} \approx 1.41, \quad UCT(n_C) = \frac{1.5}{3} + \sqrt{2}\sqrt{\frac{\ln(7)}{3}} \approx 1.64$$

Since $UCT(n_C) > UCT(n_B)$, we follow node n_C.

ii. Next, we apply the UCT formula to the children of n_C.

$$UCT(n_F) = \frac{0.3}{1} + \sqrt{2}\sqrt{\frac{\ln(4)}{1}} \approx 1.97, \quad UCT(n_G) = \frac{0.7}{1} + \sqrt{2}\sqrt{\frac{\ln(4)}{1}} \approx 2.37$$

Since $UCT(n_G) > UCT(n_F)$, we follow node n_G.

iii. Since n_G has no children, the selection phase ends here.

Expansion phase. From n_G, two possible actions are available: the dispatching rules FIFO and SPT, leading respectively to nodes n_J and n_K. We select one at random, which leads in our case to n_K. The chosen action (SPT) is applied, meaning that operation 8 is scheduled at the earliest available time on machine II.

Simulation phase. At node n_K, we perform a simulation. The remaining unscheduled operations (indicated by hatching in the figure) are randomly scheduled N_{sim} times, while respecting all precedence constraints. For each simulation, the resulting schedule is evaluated and a reward is computed. Among these simulations, the best reward is retained; in this example, it equals 0.5.

Backpropagation phase. Finally, the statistics of node n_K and all its ancestors (n_A, n_C and n_G) are updated (Table 1). For each visited node n_X, the visit count N_X is incremented by one and the cumulative reward W_X is increased by the best reward obtained during the simulation.

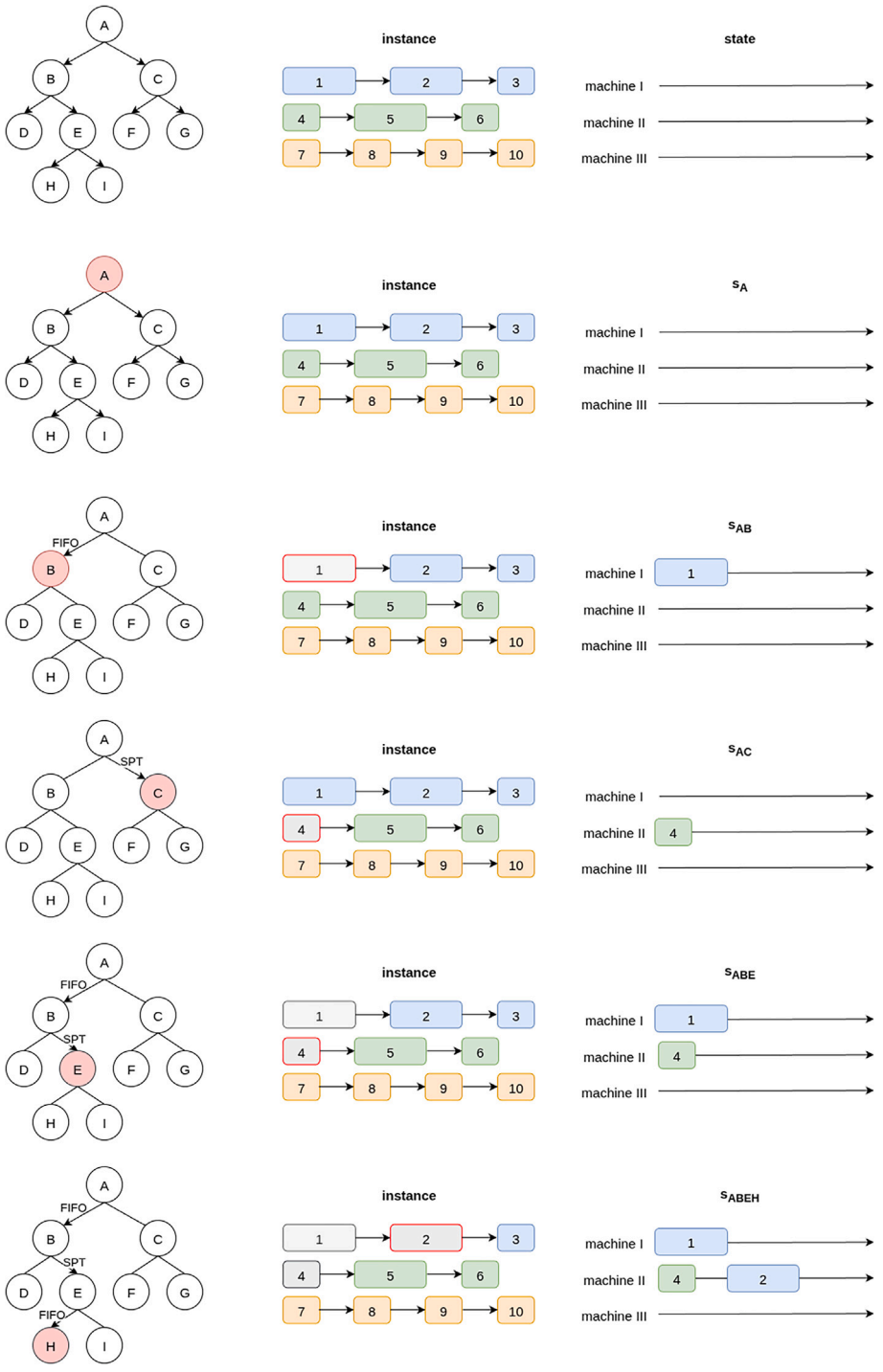

Fig. 3. Illustration of the partial search tree before the iteration, where nodes correspond to states and edges correspond to actions defined by PDRs.

3.5. Constraint programming

Constraint programming is a declarative programming paradigm for modelling and solving combinatorial problems. By integrating constraint programming into our analysis, we can compare its performance with the MCTS approach. This method is more sophisticated and involves a longer computational process than PDRs. As MCTS typically requires considerable computing time to converge on good solutions, the use of constraint programming allows for a fairer comparison of the results.

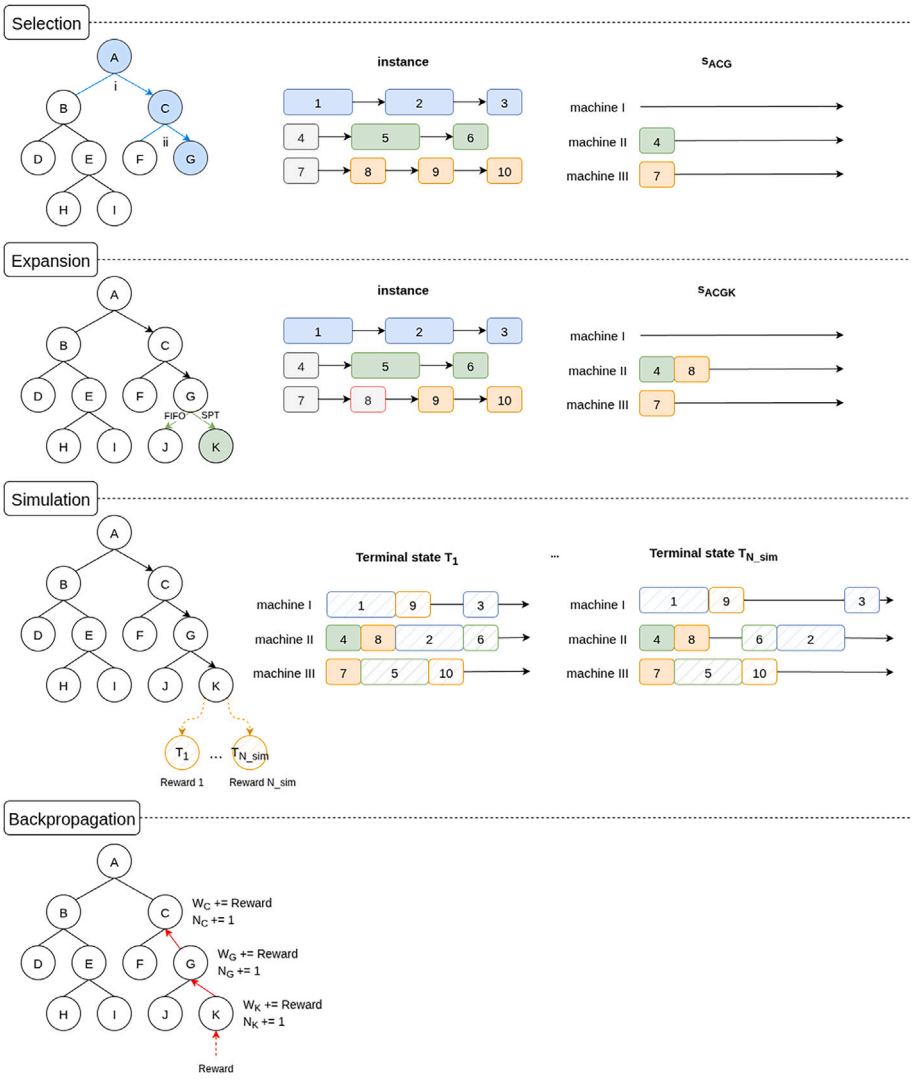

Fig. 4. One complete iteration of MCTS on a toy example, showing the four phases: selection, expansion, simulation and backpropagation.

Our CP model follows the standard job-shop scheduling formulation of Da Col and Teppan [17] and is implemented using the OR-Tools CP-SAT solver [18].

Variables

- $start_{ij}$: start time of operation O_{ij}.

Data

- p_{ij}: processing time of operation O_{ij}.
- w_i: weight of job J_i.
- $M_{(ij)}$: machine required for operation O_{ij}.

Objective

Minimize the total weighted completion time of the jobs:

$$\min \sum_{O_{ij} \text{ is last in } J_i} w_i \cdot \left(start_{ij} + p_{ij}\right).$$

Constraints

- **Precedence constraints:** For each job $J_i \in \mathcal{J}$:

$$start_{i,j+1} \geq start_{ij} + p_{ij}, \quad j = 1, \ldots, n_i - 1.$$

- **Machine capacity constraints:** For any two distinct operations $O_{ij} \neq O_{i'j'}$ assigned to the same machine $M_{(ij)} = M_{(i'j')}$:

$$start_{ij} < start_{i'j'} \rightarrow start_{i'j'} \geq start_{ij} + p_{ij}.$$

We encode each operation as an interval variable and enforce machine exclusivity through `NoOverlap` constraints available in the OR-Tools CP-SAT solver. Along with the constraint programming model, we guide the search process by using the SPT PDR, which helps refine the search strategy and improve the efficiency of finding solutions.

4. Data generation

There is a gap in the existing literature regarding job shop scheduling benchmarks. Most commonly referenced instances, such as those proposed by Taillard et al. [15], Adams et al. [3] or Demirkol et al. [19], focus on small and rectangular configurations where the number of machines equals the number of operations for each job. This structure does not adequately represent the complexities of larger, unbalanced scenarios commonly encountered in real-world manufacturing.

To address this gap, we first analyzed an industrial dataset from a real manufacturing facility that includes 51 machines, 828 jobs and a total of 6057 operations. In this instance, the workload distribution is unbalanced, with some machines heavily loaded while others are lightly used. Furthermore, the number of operations per job varies significantly, ranging from 1 to 20.

Using the statistical properties of this dataset (e.g., job type frequencies, machine type distributions, operation counts and processing times), we then generated a synthetic job shop scheduling benchmark. This approach preserves the complexity of the original environment while introducing variability and noise to allow for multiple realistic test scenarios. The data generation process is as follows:

1. Job configuration: we first generate a random integer between 600 and 1000 to determine the number of jobs. A type and a size are assigned to each job. We have two different types of jobs: common and unique. Common jobs are job types that occur more frequently in the job shop scheduling environment. They correspond to more frequent sets of pieces to manufacture. Unique jobs are job types that occur less frequently. They correspond to unique orders a manufacturing industry can receive. The sizes are drawn from a Gaussian distribution with a mean and standard deviation derived from the original instance.
2. Machine configuration: we first generate a random integer between 50 and 70 to define the number of machines. They are then split into different types based on their operational characteristics. A specific distribution of the number of operations is assigned to each type. Once the machine types are identified, we introduce noise to the probability distribution of these types. The number of machines of each type is then determined by sampling from this noisy distribution and their number of operations is drawn.
3. Operations to jobs assignment: with machine types and their distributions defined, we assign operations to jobs. Each job's operations are distributed across the available machines based on the machine type distribution. We also ensure that the processing times for each operation are generated from a normal distribution centred around the mean processing time for the machine type, with a specified standard deviation.

By following this process, we generated 20 instances of the JSSP based on the original data.[2] These instances are used to evaluate the performance of the MCTS algorithm in different scenarios.

5. Experiments

5.1. Setup

In this work, we evaluate the performance of five different types of environments for the MCTS algorithm. Each of these types of environments has three possible actions in its action space, except for the fourth type, which has six actions. The types of action spaces and the corresponding set of dispatching rules and, when applicable, the percentages are listed in Table 2.

We note that we encountered significant computational problems due to the long-running time when evaluating the performance of the MCTS algorithm in environment Type 3. Specifically, the number of operations is high, making the search process computationally expensive. One of the key issues arises from the need to recompute the completion times of all operations to identify the idle time at each new state. This means that at each newly scheduled operation, we recompute all the completion times. As a result, this type of environment is not feasible for our use.

[2] https://github.com/LaurieBvrx/large-scale-complex-JSSP-benchmark.git.

Table 2
Summary of environment types and characteristics for MCTS algorithm evaluation.

Env number	Type of action	PDR and p
1.1	1	{FIFO, LWR, MWR}
1.2	1	{FIFO, LOR, MOR}
1.3	1	{FIFO, SPT, LPT}
1.4	1	{LWR, LOR, SPT}
2.1	2	{FIFO, SJF, LJF}
2.2	2	{FIFO, LWF, MWF}
2.3	2	{FIFO, SJF, LWF}
3	3	*Not applicable due to high computational cost*
4.1	4	{LWF}, [0.6, 0.8, 1.0]
4.2	4	{LWF}, [0.3, 0.6, 0.8]
4.3	4	{MWF}, [0.6, 0.8, 1.0]
4.4	4	{MWF}, [0.3, 0.6, 0.8]
4.5	4	{SJF}, [0.6, 0.8, 1.0]
4.6	4	{SJF}, [0.3, 0.6, 0.8]
4.7	4	{LJF}, [0.6, 0.8, 1.0]
4.8	4	{LJF}, [0.3, 0.6, 0.8]
5.1	4	{LWF, MWF}, [0.6, 0.8, 1.0]
5.2	4	{LWF, MWF}, [0.3, 0.6, 0.8]
5.3	4	{SJF, LJF}, [0.6, 0.8, 1.0]
5.4	4	{SJF, LJF}, [0.3, 0.6, 0.8]
5.5	4	{LWF, SJF}, [0.6, 0.8, 1.0]
5.6	4	{LWF, SJF}, [0.3, 0.6, 0.8]

MCTS configuration. We use $N_{\text{repeat}} = 6$ iterations and $N_{\text{sim}} = 30$ simulations per root update. The number of root updates depends on the environment:

$$N_{\text{rootUpdate}} = \begin{cases} \text{number of operations}, & \text{env. type 1}, \\ \text{number of jobs}, & \text{env. types 2, 4, 5}. \end{cases}$$

The results are compared with the constraint programming model Section 3.5. All the algorithms are coded in Python and the constraint programming model is solved using the Google OR-Tools library [18]. The computations are executed on a calculation server with 48 Intel Xeon E7 v4 2.20 GHz processors and a total RAM of 128 GB.

5.2. Results

Using performance profiles, we explore the different configurations with the same environment type and then compare the best configurations across all environment types. A performance profile represents the percentage y of instances for which a specific method produces a solution whose objective function is no worse than x times the best solution found by any of the studied methods.

Before analyzing the environments in detail, we compare the performance of all PDRs used across the different environments in Fig. 5. The results reveal significant differences between the rules, with performance scores ranging from almost twice as good for the best PDRs compared to the weakest ones. Among them, LWF and LWR overlap almost perfectly and clearly dominate the other rules. This behaviour is expected: selecting the operation of the job with the least work remaining implicitly tends to schedule the entire job in one go, since after processing one operation, the job often remains the one with the least work remaining. This mechanism makes LWR behave similarly to LWF, apart from slight variations when ties occur between two jobs at a given time.

Having established the comparison of the PDRs in isolation, we now examine how they perform when embedded within different environmental configurations. Each figure in the following presents the performance profiles of all configurations within one environment type, as summarized in Table 2.

In Fig. 6, the performance profiles indicate that Configuration 1.4 consistently outperforms all other configurations of Type 1. Fig. 7 evaluates the configurations of Type 2. Configuration 2.3 dominates the other configurations, achieving the best performance across all instances for the configuration Type 2. Fig. 8 presents the results for configurations of Type 4. Configurations 4.1 and 4.2 achieve the best performance, dominating the other configurations. Configurations with the same PDR but different percentages have similar performance, indicating that the PDR is a key determining factor. The results suggest that configurations using the LWF PDR consistently perform better, followed by those with SJF, MWF and finally LJF. Fig. 9 compares configurations of Type 5, which demonstrate a similar pattern to Type 4. Configurations 5.6 and 5.7 outperform the other configurations. Again, configurations with the same pair of PDRs but different percentages perform equivalently. Configurations with LWF and SJF outperform those using LWF and MWF, which in turn perform better than those using SJF and LJF.

Additionally, the mean performance across all instances for each configuration, is presented in Table 3.

Finally, we compare the best-performing configurations of Type 1, 2, 4, and 5 alongside the constraint programming results and the best PDR LWF. Fig. 10 gives the resulting performance profiles. We observe that configurations from Type 4, 5 and 2 achieve the best performance for 50 %, 40 % and 10 % of the instances, respectively. This highlights the benefit of using a less greedy and more flexible environment, as seen in Type 4 and 5, which process operations during idle time even if the operation processing time

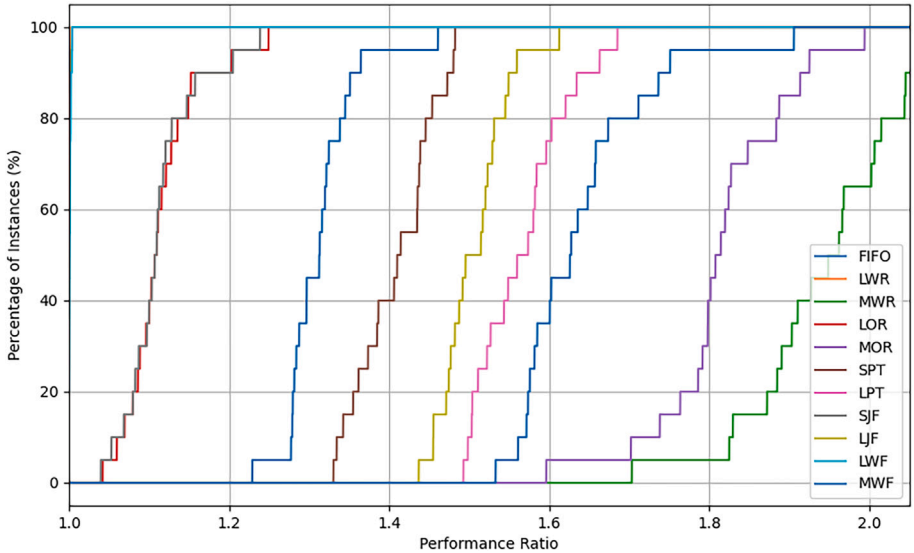

Fig. 5. Performance profiles of the PDRs.

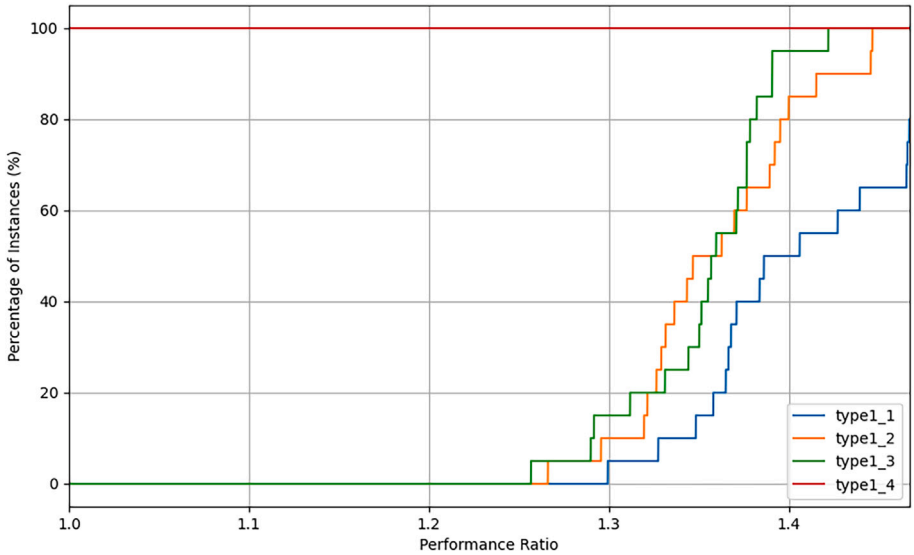

Fig. 6. Performance profiles of the configurations of Type 1.

exceeds the available time. Additionally, the results indicate that the MCTS-based algorithm outperforms the CP approach on the large instances we have considered, even when the latter is paired with a search heuristic. Interestingly, the best Type 1 configuration does not outperform the standalone LWR dispatching rule. We attribute this to the large search depth (≈ 4000–6000 decision steps). Under our budget ($N_{repeat} = 6$, $N_{sim} = 30$) MCTS cannot propagate value estimates effectively. In addition to solution quality, we also analyze the computational effort required by the different environment types. Fig. 11 reports the average runtime per instance size across the different types of environments. The results indicate that Type 1 is the least scalable, as it relies on operation-level decisions that require deeper trees.

In contrast, Type 2 is the most scalable, since it makes decisions at the job level and avoids recomputing completion times at each step. Type 4 and 5 lie in between, offering a trade-off between computational effort and solution quality. The results show that these configurations can lead to improvements in objective value, with reductions of about 5 %–6 % compared to LWF (see Table 3). However, these gains come at the expense of runtime: while a simple PDR can solve instances almost immediately, Type 4 and 5 may require between 1 h and three and a half hours. Thus, if maximizing solution quality is critical for the application, environments 4 or 5 should be preferred; otherwise, a simpler PDR remains the more practical choice. It is also worth noting that scheduling is often performed in an offline setting, where decisions do not need to be made in real time. In such cases, the additional computational effort required by environments 4 and 5 can be justified.

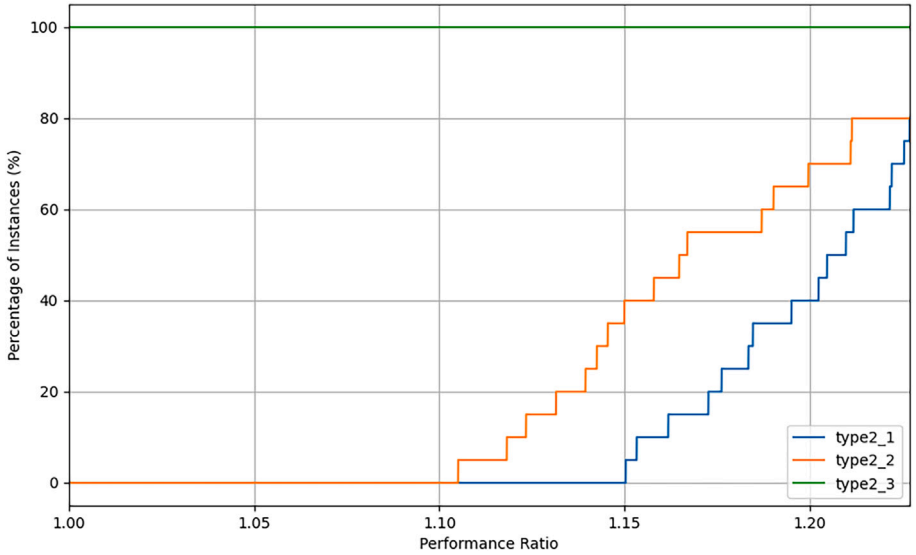

Fig. 7. Performance profiles of the configurations of Type 2.

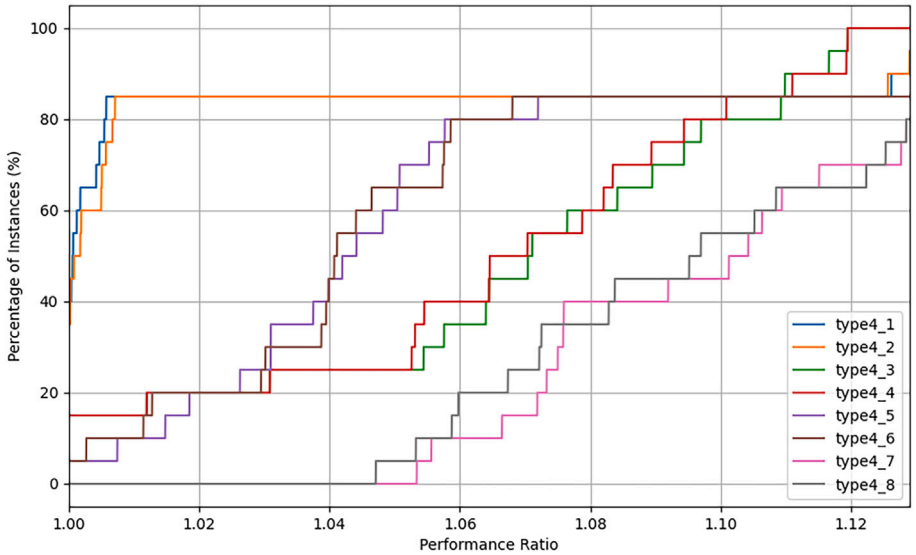

Fig. 8. Performance profiles of configurations of Type 4.

6. Conclusion

In this study, we explore the potential of using MCTS to solve large-scale and real-world inspired instances of the JSSP. We introduce various MDP formulations to model the JSSP for the MCTS algorithm and compare their performance with a constraint programming model. In addition, we deliver a new synthetic benchmark derived from anonymised real-world manufacturing data that captures the complexity and variability of industrial scheduling environments. Our experimental results show that MCTS is a promising approach for solving large-scale JSSPs, consistently outperforming our constraint programming approach. The MCTS-based algorithm shows better performance in different MDP formulations. In particular, configurations that allow operations to be inserted in idle time shorter than their processing time are beneficial. The proposed MDP formulations provide a flexible framework for representing different scheduling problems and the new benchmark is a valuable tool for testing and evaluating scheduling algorithms in industrial contexts.

Despite these encouraging results, several limitations remain. First, the computational cost of MCTS can still be high for very large instances, especially when operation-level actions are used, which limits scalability. Second, the quality of the solutions depends on the choice of parameters (e.g., tree depth, exploration–exploitation balance), which may require extensive tuning for different problem instances. Finally, our current implementation is purely sequential and does not exploit parallel processing capabilities.

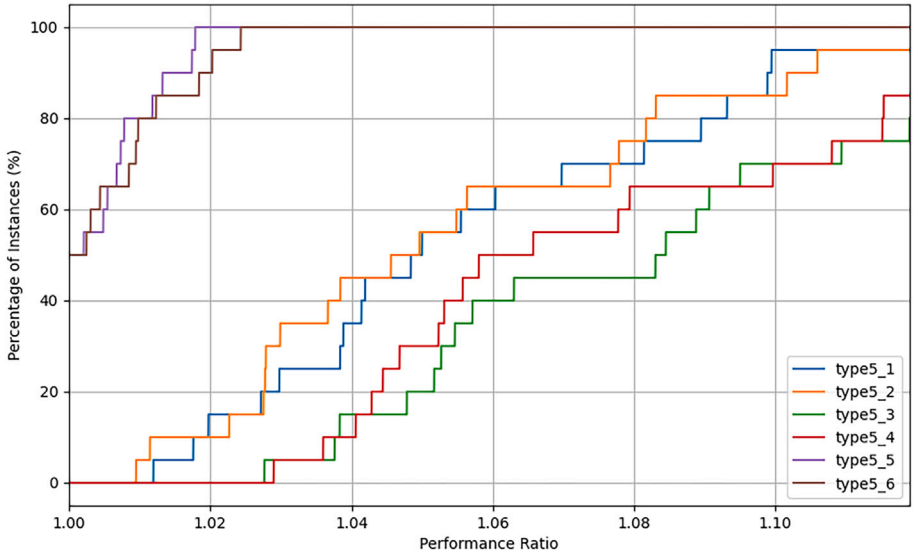

Fig. 9. Performance profiles of the configurations of Type 5.

Table 3
Mean of the weighted sum of the completion times (scaled down by a factor of 10^8) across all instances of each configuration of the MCTS and of the constraint programming approach. Bold values indicate the best performance (lowest mean) among each type of MCTS configuration.

Method	Mean
Constraint programming	1.7136
LWF	1.6456
MCTS - Type 1.1	2.4014
MCTS - Type 1.2	2.2852
MCTS - Type 1.3	2.2859
MCTS - Type 1.4	**1.6837**
MCTS - Type 2.1	2.0226
MCTS - Type 2.2	1.9875
MCTS - Type 2.3	**1.6792**
MCTS - Type 4.1	**1.5532**
MCTS - Type 4.2	1.5533
MCTS - Type 4.3	1.5815
MCTS - Type 4.4	1.5786
MCTS - Type 4.5	1.6234
MCTS - Type 4.6	1.6225
MCTS - Type 4.7	1.6586
MCTS - Type 4.8	1.6552
MCTS - Type 5.1	1.6312
MCTS - Type 5.2	1.6249
MCTS - Type 5.3	1.6763
MCTS - Type 5.4	1.6644
MCTS - Type 5.5	**1.5415**
MCTS - Type 5.6	1.5437

However, parallelization of MCTS is readily achievable and has been shown to significantly reduce runtime while maintaining or even improving the solution quality. There exist methods for parallelizing MCTS [20], including:

- Leaf parallelization, where multiple simulations are performed in parallel from the same node.
- Root parallelization, where multiple independent MCTS trees are built in parallel and their root statistics are merged.
- Tree parallelization, which allows concurrent traversal of the same tree using techniques like mutexes or virtual losses to manage shared state .

Incorporating such techniques could therefore significantly reduce runtime and further improve the performance of our approach, making it more practical for even larger scheduling problems.

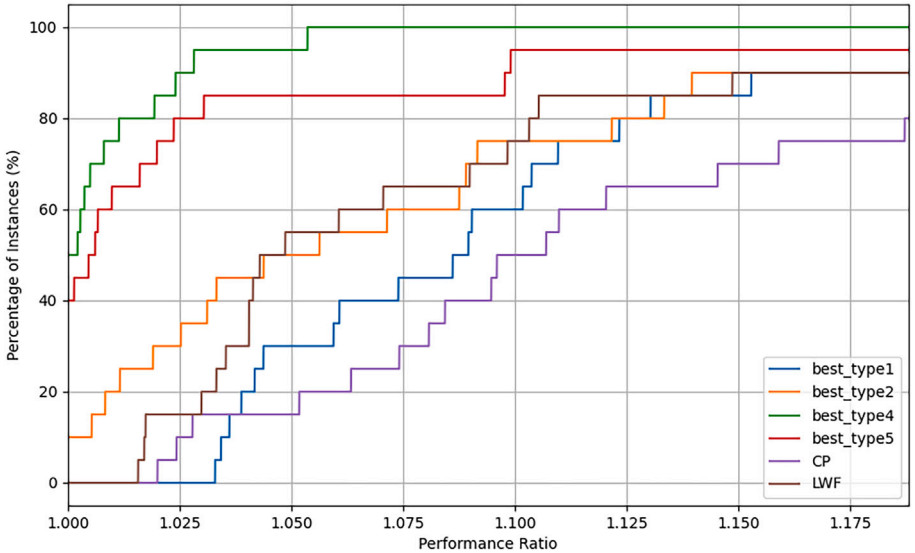

Fig. 10. Performance profiles comparing the best-performing configuration of Type 1, 2, 4 and 5, along with the constraint programming and the best PDR results.

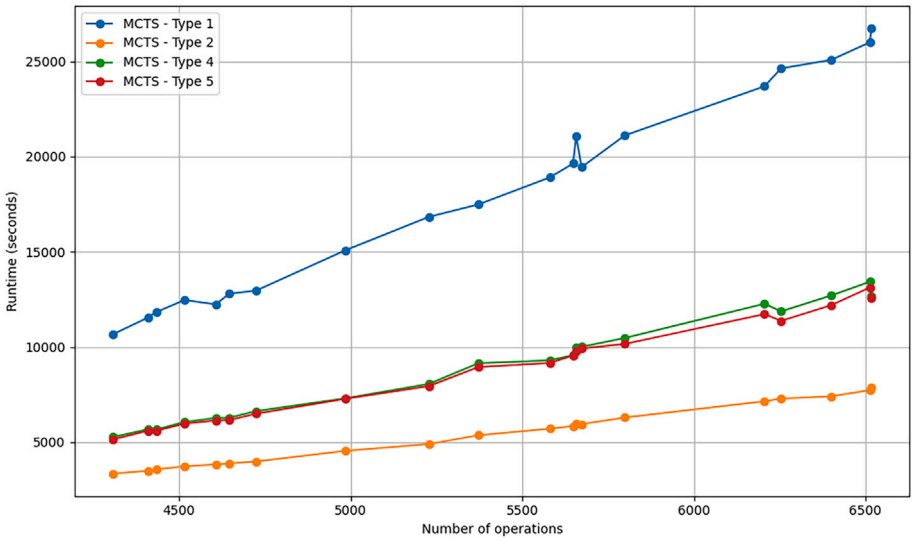

Fig. 11. Average runtime per instance size across environments.

Future research could further refine the MCTS approach by exploring a machine-learning-based reward function allowing the evaluation of a partial schedule. In conclusion, our results support the utility of MCTS as an alternative heuristic to solve large job shop scheduling problems.

CRediT authorship contribution statement

Laurie Boveroux: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Damien Ernst:** Writing – review & editing, Supervision, Conceptualization. **Quentin Louveaux:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Conceptualization.

References

[1] M.L. Pinedo, Modeling and solving scheduling problems in practice, in: Scheduling: Theory, Algorithms, and Systems, Springer, 2012, pp. 431–458.
[2] M.R. Garey, D.S. Johnson, R. Sethi, The complexity of flowshop and jobshop scheduling, Math. Oper. Res. 1 (2) (1976) 117–129.
[3] J. Adams, E. Balas, D. Zawack, The shifting bottleneck procedure for job shop scheduling, Manag. Sci. 34 (3) (1988) 391–401.
[4] J. Carlier, The one-machine sequencing problem, Eur. J. Oper. Res. 11 (1) (1982) 42–47.
[5] D.C. Hajariwala, S.S. Patil, S.M. Patil, A review of metaheuristic algorithms for job shop scheduling, Eng. Access 11 (1) (2025) 65–91.

[6] C. Zhang, W. Song, Z. Cao, J. Zhang, P.S. Tan, X. Chi, Learning to dispatch for job shop scheduling via deep reinforcement learning, Adv. Neural Inf. Process. Syst. 33 (2020) 1621–1632.
[7] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of go with deep neural networks and tree search, Nature 529 (7587) (2016) 484–489.
[8] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Trans. Comput. Intell. AI Games 4 (1) (2012) 1–43.
[9] T.P. Runarsson, M. Schoenauer, M. Sebag, Pilot, rollout and Monte Carlo tree search methods for job shop scheduling, in: International Conference on Learning and Intelligent Optimization, Springer, 2012, pp. 160–174.
[10] J.-J. Chou, C.-C. Liang, H.-C. Wu, I.-C. Wu, T.-Y. Wu, A new MCTS-based algorithm for multi-objective flexible job shop scheduling problem, in: 2015 Conference on Technologies and Applications of Artificial Intelligence (TAAI), IEEE, 2015, pp. 136–141.
[11] T.-Y. Wu, I.-C. Wu, C.-C. Liang, Multi-objective flexible job shop scheduling problem based on Monte-Carlo tree search, in: 2013 Conference on Technologies and Applications of Artificial Intelligence, IEEE, 2013, pp. 73–78.
[12] M. Lubosch, M. Kunath, H. Winkler, Industrial scheduling with Monte Carlo tree search and machine learning, Procedia CIRP 72 (2018) 1283–1287.
[13] M. Saqlain, S. Ali, J.Y. Lee, A Monte-Carlo tree search algorithm for the flexible job-shop scheduling in manufacturing systems, Flex. Serv. Manuf. J. 35 (2) (2023) 548–571.
[14] M. Loth, M. Sebag, Y. Hamadi, M. Schoenauer, C. Schulte, Hybridizing constraint programming and Monte-Carlo tree search: application to the job shop problem, in: International Conference on Learning and Intelligent Optimization, Springer, 2013, pp. 315–320.
[15] E. Taillard, Benchmarks for basic scheduling problems, Eur. J. Oper. Res. 64 (2) (1993) 278–285.
[16] R. Reijnen, K. van Straaten, Z. Bukhsh, Y. Zhang, Job shop scheduling benchmark: environments and instances for learning and non-learning methods. arXiv preprint arXiv:2308.12794, 2023.
[17] G. Da Col, E.C. Teppan, Industrial-size job shop scheduling with constraint programming, Oper. Res. Perspect. 9 (2022) 100249.
[18] L. Perron, F. Didier, Cp-sat, Feb 2025. https://developers.google.com/optimization/cp/cp_solver/. Version v9.12, Google.
[19] E. Demirkol, S. Mehta, R. Uzsoy, Benchmarks for shop scheduling problems, Eur. J. Oper. Res. 109 (1) (1998) 137–141.
[20] G.M.B. Chaslot, M.H. Winands, H.J. Van Den Herik, Parallel Monte-Carlo tree search, in: Proceedings of the International Conference on Computers and Games, Springer, 2008, pp. 60–71.