Politecnico di Torino

Corso di Laurea Magistrale in

Ingegneria Matematica

Reinforcement Learning for Dynamic Stochastic Scheduling

Relatore:
Paolo Brandimarte

Candidato:
Alessia De Crescenzo

Correlatore:
Edoardo Fadda

Anno Accademico 2023/2024

Abstract

With the sharp increase of uncertainty and complexity in production processes, dynamic scheduling nowadays plays a strong role in making enterprises more competitive: it is needed to handle real time events, such as machine breakdowns, job arrivals and stochastic processing times.
The static job scheduling problem (JSP) is one of the most practically relevant but rather complex scheduling problems, having been proved to be NP hard, and it has been the subject of a significant amount of literature in the operations research field: however, this approach is unrealistic in real-world contexts,where dynamic events such as insertions, cancellations or modifications of orders, machine breakdowns, variation in due dates and processing times are inevitable and drive the realized execution of a static schedule far from its expected outcome and deteriorate the production efficiency seriously.

This work focuses on dynamic scheduling problem in job shops with new job arrivals at stochastic times, aiming at minimizing the penalties for earliness, tardiness and flowtime, according to the just-in-time (JIT) policy, which is based on the idea that early as well as late delivery must be discouraged: a Reinforcement Learning agent-based method for developing a predictive-reactive scheduling strategy is investigated.

The approach involves generating an initial schedule and subsequently revising it in response to the arrival of new jobs. Specifically, the proposed method entails implementing an event-driven rescheduling policy, wherein the arrival of a new job prompts a rescheduling of the entire timeline from the arrival time onwards. An agent is designed to simulate time according to the current schedule and schedule the operations of the new job.

The developed method was tested on a wide range of instances and compared to a simple heuristic, namely a FIFO agent, and was able to exceed its performance on most instances.

Contents

Chapter 1

Introduction

Scheduling is the allocation of shared resources over time to competing activities. It has been the subject of a significant amount of literature in the operations research field. Emphasis has been on investigating machine scheduling problems where jobs represent activities and machines represent resources.
The job-shop scheduling problem (JSP) is a type of combinatorial optimization problem which determines how to assign a set of jobs on a set of machines to minimize or maximize a predefined objective function under certain constraints. It has been proved to be NP-hard, which means that the time required to get an optimal solution increases exponentially with the problem size.
Moreover, the process of scheduling tasks is significantly affected by unforeseen events such as new jobs arrivals. Therefore, the study of scheduling in the presence of real-time disruption (dynamic scheduling) is attracting increasing attention.
At present, the methods of solving the DFJSP are mainly heuristic and metaheuristic algorithms. Heuristics are simple and efficient, but often fall into local optima and their solution quality is poor due to greed and shortsighness. Metaheuristics improve the solution quality through parallel searching and iterative searching, but this is time consuming (Liu et al. [2023]).
An alternative approach is that of Reinforcement Learning, a branch of Machine Learning (ML) that can provide solutions to many real-world applications from artificial intelligence to operation research or control engineering, where an agent interacts with an environment by performing actions and perceiving environmental states and has to learn a 'correct behaviour' (the optimal policy) by means of a feedback rewarding signal.

In particular,RL processes have two components: a decision maker and its environment; the decision maker (the agent) observes the state of the environment at some discrete points in time (decision epochs) and makes decisions, i.e. takes actions based on the state.
The decisions made are then executed in the environment which will find itself in a new state later. As a response to the decision maker, the environment also returns a reward to the decision maker: the goal is to find an optimal way to make decisions so as to maximize the long-term cumulative rewards (Ana Esteso and Díaz-Madroñero [2023]).
The method proposed in this thesis entails implementing an event-driven rescheduling policy, wherein the arrival of a new job prompts a rescheduling of the entire timeline from the arrival time onwards. An agent is designed to simulate time according to the current schedule. Each time a machine becomes available, the agent evaluates both the next scheduled operation and the task associated with the new job on that machine. If the new job's operation is not feasible (e.g., other operations must be processed first on different machines), the agent refrains from modifying the schedule. Conversely, if the new operation is feasible, the agent determines whether to insert the new operation prior to the scheduled one, taking into account the reward in terms of penalty reduction.

Two agents were trained within the proposed approach and tested on a wide range of instances, with varying number of dynamic jobs. The results were then compared to those obtained by a simple heuristic, i.e. a FIFO agent, and the superiority of the two agents was proved, especially when working with large-scale instances.

Chapter 2

Dynamic Scheduling: Literature review

2.1 Dynamic Scheduling

Most of existing methods for solving the JSP have assumed a static manufacturing environment where the information on the shopfloor is completely known in advance, hence outputting a deterministic scheme without any modification during the entire working process. However dynamic events, such as machine breakdowns, rush orders, changes in job priorities or delays in job processing, are inevitable and unpredictable and can destabilize the production system and invalidate the current scheduling scheme. Therefore, it is of remarkable importance to develop new scheduling methods for Dynamic Scheduling.

2.2 A Dynamic Scheduling Framework

Before diving deep into dynamic scheduling methods, we present a brief summary of a framework for understanding dynamic scheduling research.

2.2.1 Dynamic Scheduling Strategies

According to the type of the initial schedule, dynamic scheduling strategies can be classified into three categories:

- *Completely reactive scheduling* (also named on-line scheduling), generating no pre-schedule and completing the scheduling in real time.

- *Predictive-reactive scheduling* generating a predictive schedule advance to optimize shop performance without considering possible future disruptions and adapting the schedule to the dynamic events.
- *Robust pro-active scheduling*, producing a schedule in advance to anticipate the effect of disturbance on manufacturing system.

Obviously, when using priority dispatching rules or heuristics, schedules are easily constructed in real time: they are a form of completely reactive scheduling. On the other hand, most metaheuristics belong to the second category, that of predictive-reactive scheduling, where the schedule is adjusted when a dynamic event occurs.

2.3 Dynamic Scheduling Policies

Dynamic scheduling policies are needed to implement the above strategies. The different types of existing policies are the following:

- A *periodic rescheduling policy* performs a periodical rescheduling and carries out these schedules in a rolling time range.
- In an *event-driven rescheduling policy*, a real time event occurs accompanied by a rescheduling.
- A *hybrid rescheduling policy* performs periodical rescheduling whenever a real time event occurs.

Baykasoğlu and Karaslan [2017] developed a GRASP-based approach for the DJSP and were able to show that, usually, event-driven policies outperform periodic rescheduling policies for the problems studied in their work.

2.4 Dynamic Scheduling Methods and Approaches

As an attractive research field in both academia and industry, dynamic scheduling has been intensively researched over the past decades. Various methods have been presented, the most widely used among which are dispatching rules and metaheuristics.

Dispatching rules immediately react to dynamic events, thus achieving the best time efficiency. However, they fail to guarantee even a local optimum, much less a global optimum. Meanwhile, since different rules are

suitable for different scenarios, it is hard for the decision maker to select the best rule at a specific time point.

However, they are a powerful tool: Lawrence and Sewell [1997] found that simple dispatching rules could provide comparable or even superior performance to optimum seeking methods when the uncertainty in processing times increase. Moreover, Rajendran and Holthaus [1999] conducted a comparative study on the performance of different dispatching rules in the dynamic flow shops and job shops, concluding that the performance of these rules depends on routing of jobs and shopfloor configuration.

Dynamic or real time dispatching rules have been studied and implemented: Morady Gohareh and Mansouri [2022] proposed a framework to generate dynamic and global dispatching rules for the DJSP. The generated rules have two pillars: a memory measure (pheromones,generated by simulation and ant colony algorithm) and a heuristic measure, based on the Central Limit Theorem.

Meta-heuristics are able to provide near-optimal solutions with acceptable gaps at the cost of an higher (but reasonable) computational time. Most metaheuristics proceed in an iterative search process: starting from an initial solution or a population of initial solutions, the algorithm improves the solution(s) while trying to avoid local optima via perturbation mechanisms.

These methods have been widely applied to the Dynamic Job shop: Kundakcı and Kulak [2016] introduced efficient hybrid Genetic Algorithm methodologies for minimizing the makespan, integrating GA with different heuristics to generate the initial population. They compared the proposed method with some of the main state-of-art approaches and proved that it was able to overcome them on a various range of instances.

Wang et al. [2019] proposed a multi-restart Particle Swarm Optimization algorithm in a multi-objective DJSP setting: their method allows the restart solution to be generated from a group of solutions drawn from local optima. This extends the search space, while maintaining the quality of the restart solution. When compared with six state of the art metaheuristic their approach proved to outperform five out of six and to be comparable to the last one.

To find the best order for operations to be processed, the JSP can be regarded as a Markov decision process (MDP), where an intelligent agent should determine the optimal action after the occurrence of a new arrival, by comprehensively utilizing the information form current production state (Puterman [1994]). In recent years, Reinforcement Learning (RL) has emerged

as a powerful way to deal with MDPs and has been applied to different kinds of dynamic scheduling problems.

Wang et al. [2019] present an interesting approach in their study where they simulate the dynamic arrival of jobs in a discrete event simulation (DES) system. In their system, both the arrival times of jobs and their processing times are generated from an exponential and a uniform distribution respectively. This results in each job being different, leading to a potentially infinite state space: to address this problem, they opt for an aggregate state representation.

Most RL-based methods have a "single operation" approach: at each time step, the agent decides which operation will be processed next on which machine. Clearly, this is an online scheduling approach, where the schedule is created in real-time and new job's operations are simply added to the machines buffers. In this work, a "multiple operations" approach is proposed: at each order arrival, the agent adjust the schedule for the entire job shop accordingly and outputs the ordered queues of each machine.

Chapter 3

Problem formulation

The $n \times m$ Just-In-Time job-shop scheduling problem can be described by a set of n jobs $(J_i)_{1 \leq i \leq n}$ which is to be processed on a set of m machines $(M_r)_{1 \leq r \leq m}$. Each job has a release date r_i, a due date d_i and tardiness, earliness and flowtime weights $w_{t_i}, w_{e_i}, w_{f_i}$.
Moreover, it needs to be processed on the machines in a given order. The processing of job J_i on machine M_r is defined as the operation O_{ir}. Operation O_{ir} requires the exclusive use of machine M_r for an uninterrupted period of time p_{ir}, its processing time.
In this study, the assumptions and constraints are as follows:

- Each machine can process only one operation at a time;
- The order of precedence of operations belonging to the same job must be followed and there are no precedence constraints among the operations of different jobs;
- The operation must be processed without interruption;
- Jobs are independent;
- An unlimited buffer between machines is assumed.

Finally, the objective function is the total penalty:

$$P_n = T_n + E_n + Fp_n$$

which is defined by the sum of weighted tardiness T_n, weighted earliness E_n and weighted flowtime penalty Fp_n, which are described respectively by the

following formulas:

$$T_n = \sum_{i=1}^{n} w_{t_i} \left(\max(0, C_i - d_i)\right)$$

$$E_n = \sum_{i=1}^{n} w_{e_i} \left(\max(0, d_i - C_i)\right)$$

$$Fp_n = \sum_{i=1}^{n} w_{f_i} \left(C_i - S_i\right)$$

where C_i represents the completion time of the i-th job and S_i represents the starting time of processing of the i-th job.
In this work we consider the *dynamic job shop scheduling* problem, where stochastic job arrival is considered as the 'stochastic' factor: after developing our initial schedule, we progressively receive new orders, characterized as described before, and we perform rescheduling of the entire job shop to account for each of these orders.

Chapter 4

Reinforcement Learning and Markov Decision Processes

4.1 Reinforcement Learning: a brief introduction

Reinforcement learning is one of the three main branches of ML techniques and can be considered as a third machine learning paradigm, alongside of supervised learning, unsupervised learning, and perhaps other paradigms as well: it was initially proposed in the early 1990s and has attracted a lot of interest from the research community since then.

Different from other machine learning approaches, Reinforcement Learning is the task of learning from interactions with the environment: a learning agent keeps interacting with its dynamic environment and chooses actions based on the received feedback, used to update its knowledge.

According to Sutton and Barto [2018], three main elements are required for the RL process: a policy, which maps the actions to the states; a reward signal, which classifies this action according to the immediate return received by the transition between states; a value function, to evaluate which actions have positive long term effects by considering not only the immediate reward of a state, but the long-run cumulative reward;
There is a fourth optional element in some RL systems, namely a model of the environment that mimics the environment's behaviour.

The agent and environment interact at each of a sequence of discrete time steps. At each time step t, the agent receives some representation of the environment's state, $S_t \in S$, where S is the set of possible states, and on that basis selects an action, $A_t \in A(St)$, where $A(S_t)$ is the set of actions

available in state S_t. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in R$, and finds itself in a new state, S_{t+1}.

At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted π_t.

Reinforcement learning methods specify how the agent changes its policy as a result of its experience. The agent's goal, roughly speaking, is to maximize the total amount of reward it receives over the long run.

The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning. In RL, rewards are in a sense primary, whereas values, as predictions of rewards, are secondary: without rewards there could be no values, and the only purpose of estimating values is to achieve more reward.

Nevertheless, it is values with which we are most concerned when making and evaluating decisions: we seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run.

4.1.1 Value Functions

In reinforcement learning, the purpose of the agent is formalized in terms of a reward signal passing from the environment to the agent. At each time step, the reward is a simple number, $R_t \in R$ and the agent goal is to maximize the cumulative reward obtained from its actions.

How might this be defined formally? If the sequence of rewards received after time step t is denoted $R_{t+1}, R_{t+2}, R_{t+3}, ...$ what precise aspect of this sequence do we wish to maximize? In general, we seek to maximize the expected return, where the return G_t is defined as some specific function of the reward sequence, such as the sum of the rewards.

The latter could be used in episodic tasks, where the agent reaches a terminal state after a certain number of timesteps, but it cannot be used when dealing with interactions that have no time limits: this is why discounting was introduced. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized.

In particular, it chooses A_t to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{4.1}$$

where $\gamma, 0 \leq \gamma \leq 1$, is called the discount rate.

Almost all reinforcement learning algorithms involve estimating value functions, functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state), in terms of expected future return. Clearly, the rewards the agent can expect in the future depend on the chosen actions: therefore, value functions can be defined with respect to particular policies.

Recalling that a policy is a mapping from each state-action couple (s,a) to the probability of selecting action a when in state s, the value of a state s under policy π, denoted $v_\pi(s)$, can be defined as the expected return starting from s and following π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \tag{4.2}$$

where γ is the discount rate and $\mathbb{E}_\pi[\cdot]$ denotes the expectation under policy π.

Similarly, the value of taking action a in state s under policy π, denoted $Q_\pi(s, a)$, is the expected return starting from s, taking action a, and following π:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{4.3}$$

These functions can be estimated from experience: if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. If separate averages are kept for each action taken in a state, then these averages will similarly converge to the action values, $q_\pi(s, a)$. Estimation methods of this kind are called Monte Carlo methods because they involve averaging over many random samples of actual returns.

Of course, if there is an high number of states, it may not be possible to store separated averages for each state, or for each state-action pair: the agent can keep v_π and Q_π as parameterized functions and adjust the parameters to better match the observed returns.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy the Bellman's equation:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]$$

$$= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s\right]$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_{t+1} = s'\right]\right]$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right], \tag{4.4}$$

for any policy π and any state s.

It expresses a relationship between the value of a state and the values of its successor states: starting from state s, the agent could take any of some set of actions; the Bellman equation averages over all possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.

The value function v_π is the unique solution to its Bellman equation. On the other hand, this equation forms the basis of a number of ways to compute, approximate, and learn v_π.

4.2 Markov Decision Processes Modelling

RL processes can be modeled as a Markov decision process (MDP): this stochastic mathematical model can be described by a five-tuple

$$MDP = \langle S, A(s), P(s'|s,a), \gamma, R(s'|s,a) \rangle$$

where S is a set of all possible states of the environment; A(s) is a set of possible actions (alternatives) while the state is s, $s \in S$; $P(s'|s,a)$ is a set of probabilities that the state of the environment changes from state s to state s' after action a is taken; γ is a discount factor; $R(s'|s,a)$ is a set of rewards that the decision maker obtains after taking action a and the state of the environment changes from state s to state s'(Zhang et al. [2017]).
It is assumed in this work that the only stochastic factor of the job shop is represented by the arrival of different orders: whenever a new order is received, the agent needs to make a decision on how to insert it in the schedule. In particular, we assume job inter-arrival times as exponentially distributed with mean $\frac{1}{\lambda}$. Therefore, the arrival of orders can be modeled as a birth chain (a Poisson process).
Moreover, we suppose that the job shop includes a buffer space to accommodate newly arrived jobs with a capacity of 1, meaning it can hold at most one job at a time. If a job arrives when the buffer is empty, it can be set to wait until the next instant of time to be either scheduled or discarded. However, if a job arrives when the buffer is already occupied, it cannot enter the buffer and is immediately discarded.
Finally, the state of the system is observed at each instant of time but actions are allowed only when a new job enters the factory.

4.2.1 Decision epochs and non-decision epochs

Decision are made only when a job arrives: we call these points of time decision epochs. Contrarily, no decisions are made during the remaining time instants and we refer to these moments as non-decision epochs.
To model this, we decide to construct different set of actions depending on the state and, in particular, we defined the set of action in non-decision epochs as containing only the action "Don't change the schedule".

4.2.2 State representation

In order to model this decision problem as a *MDP*, we first need to define the system state-space: the state represents the current configuration of the

job shop and must be chosen such that the Markovian property holds and therefore the state changes only depend on the present state and the chosen action.
At each epoch, the system state can be denoted by the vector

$$S_t = (O_t, \eta_t, \mu_t, \theta_{1,t}, ..., \theta_{m,t})$$

where

- O_t takes value equal to 1 if the buffer is full (i.e. a new order is available) and 0 otherwise.
- η_t is a list of integers, each representing the number of completed operations for the i-th job.
- μ_t is a list of lists containing the ordered queues for each machine.
- θ_t is a dataframe, containing the current schedule for the job shop up to time t.

4.2.3 Action representation

The action space set A is defined as the set of all possible action that can be taken by the agent; however, in the considered framework, the decision maker takes an action only when a new arrival occurs. Therefore, we can set $A_t = \{"Do\ not\ modify\ the\ schedule"\}\ \forall t \mid O_t = 0$.
For the remaining time instants, the chosen action will be represented by a list of lists

$$A_t = (a_1, ..., a_m)$$

where a_j corresponds to the ordered queue to be processed by the j-th machine from the time of the job arrival up to the end of the schedule.

4.2.4 Transition Probabilities

Since the action space doesn't include actions affecting the arrival of new orders, the transition probabilities won't directly depend on the chosen action: the latter will deterministically affect the schedule of the job shop. Moreover, the transition probabilities will simply depend on the underlying birth process of new arrivals.

Therefore, we can define our transition probabilities in the following manner:

$$\mathbb{P}\left((0, \eta_{t+1}, \mu_{t+1}, \theta_{t+1}) \,|\, (O_t, \eta_t, \mu_t, \theta_t), A_t\right) =$$
$$\mathbb{P}\left(No\ \ arrivals\ \ observed\ \ in\ \ [t, t+1]\right) = e^{-\lambda(t+1-t)} = e^{-\lambda}.$$

$$\mathbb{P}\left((1, \eta_{t+1}, \mu_{t+1}, \theta_{t+1}) \,|\, (O_t, \eta_t, \mu_t, \theta_t), A_t\right) =$$
$$\mathbb{P}\left(At\ \ least\ \ one\ \ arrival\ \ observed\ \ in\ \ [t, t+1]\right) = 1 - e^{-\lambda(t+1-t)} = 1 - e^{-\lambda}.$$

4.2.5 Reward function

Remembering that our objective is to reduce the total weighted penalty given by tardiness, earliness and flowtime costs, simply defined by the following formula:

$$P_n(t) = \sum_{i=1}^{N_t} \max(w_{e_i}(d_i - C_i), w_{t_i}(C_i - d_i)) + w_{f_i}(C_i - S_i)$$

where d_i indicates the due date of job i,C_i its completion time and S_i the starting time of its processing. Then we can define the reward obtained over period from t to t+1 in the following terms:

$$r_t(S_{t+1}|S_t, A_t) = P_n(t) - P_n(t+1)$$

Note that, if $P_n(t+1) < P_n(t)$, this indicates that total penalty of the job shop decreased and the immediate reward is greater than 0, increasing the cumulative reward. If $P_n(t) = P_n(t+1)$, which is always true when the job shop didn't receive any new order, the reward will be equal to 0. Finally, if $P_n(t+1) > P_n(t)$ then the penalty of the job shop increased and therefore we'll obtain a negative reward.

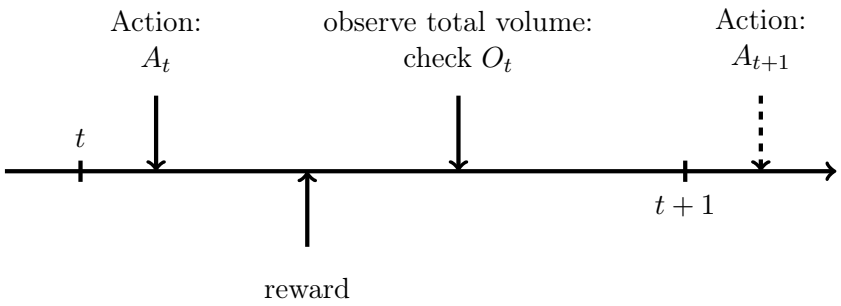


Figure 4.1: Events timeline

Chapter 5

Environment Design

A key component of this project was the implementation of the dynamic job shop environment, achieved following the structure of the Gymnasium library developed by openAI.

5.1 Environment Model

In order to present the implementation of the constructed environment, we begin by describing in more detail the simulated system considered during both the testing and training phases:

- the system is composed by m machines
- at the start of each episode, the system already presents an initial schedule for the n_s static jobs
- from the first arrival time t_a, the job shop receives n_d new orders, whose arrival follows a exponential distribution.
- each time a new job arrives, the environment refers to the agent to obtain ordered queues for each machine and then schedules the job shop based on the given queues.

Before deep diving into the implementation of these functions , it's important to note that the DJSP can be modeled and approached in various ways. The method I chose addresses the problem using a "multiple" approach, focusing on the evolution of the system one job at a time.

An alternative approach is to focus on one operation at a time: when an operation is completed on a machine, the agent's task is to select only the

next operation for that machine, repeating this process until the end of the simulation.

The methods and techniques developed for the latter are more common, and the literature is more extensive. However, the approach considered in this project ensures that the schedule does not actively depend on the agent until all jobs have been processed. Thus, in case of a malfunction of the agent, all jobs up to the last one received are scheduled and their processing can be completed.

This is an important advantage if we consider the possibility of applying this method within the context of digital twin technologies, where a replica of the real environment is generated and used to schedule the system.

5.2 Environment Implementation

The Gym framework allows the construction of various environments, simulating tasks or games in which an agent interacts with its surroundings to learn optimal behaviors.

The main methods of the environment's class are:

- *_init_* (Initialization Function): This function simply initializes the environment.
- *restart(instance name, timing, event manager)* This function initializes the environment's parameters, such as the state space, action space, and any environment-specific variables (e.g., job queues or machine statuses in a job shop). It also sets up any initial conditions required to start the simulation.
- *step(action)* (Step Function): This function takes an action as input and applies it to the environment, moving it from its current state to a new state.
- *reset()* (Reset Function): This function resets the environment to its initial state, providing a new starting point for an episode. It is typically called at the beginning of each new episode or when the environment reaches a terminal state.

Therefore, following the pseudo-code of a Gym-like environment, I implemented the methods above to simulate the considered environment.

5.2.1 Restart function

The initialization of the environment class contains some essential steps:

- reading the instance and obtain an initial solution for the first n_s jobs, using commercial solver Gurobi, therefore initializing the current schedule of the environment.
- initialize the ordered queues following the given initial schedule and, for each job, static or not, a counter for completed operations.
- initialize some functions such as timing, envmanager and reward, which will be used later on in the step function, and the list of dynamic events.

This includes the initialization of one of the fundamental elements of the whole code, which is the event list. Within this list are entered all the arrival times of the simulated dynamic jobs.

5.2.2 Step function

The step function is central to the environment's evolution. It processes the incoming action taken by the agent, ensures that this action is accurately recorded in the environment (i.e. that the system is rescheduled based on the new queues), and updates the environment's state up to the next event.

Three main operations take place within it:

- "timing" operation: given the queues produced by the agent, this function reschedules the remaining operations. This is achieved by solving a Gurobi model where precedence constraints on machines are fixed based on the action of the agent. The model is initialized once at the start of each episode and is updated each time an operation is either removed from or inserted into the job shop.
- "next" operation: this function handles job arrivals, by placing the new job in the job shop buffer and fast forwarding the environment's time up to the time of the new arrival.
- reward computation: the reward obtained by modifying the schedule and inserting the previous dynamic job is computed.

5.2.3 Reset function

The reset function is called at the start of each episode and its mainly responsible for reading and presenting to the environment the first job arrival from the events list, after the environment was restarted.

5.3 Ausiliary classes and methods

To manage the arrival of orders more accurately, the environment was integrated with an event manager and a timing structure: the former handles the events for the environment by reading, recording, and presenting them one at a time, while the latter is responsible for solving the scheduling problem using Gurobi, maintaining the order presented by the agent in each queue and taking into account setup times and precedence constraints.

5.3.1 Event manager

The event manager is initialized at each restart of the environment and independently manages the arrival of new jobs and their departure from the shop floor: specifically, it is responsible for keeping the queues, the counters of completed operations for each job, and the timing model updated and presenting each new job to the environment at the time of its arrival.

When initialized, it start by reading the list of dynamic jobs to be received by the job shop and inserts each element in a list, together with its arrival time.

The main method of this class is the "next" method: this is called by the environment at each step and at each reset and starts by reading the next arrival from the events list and inserting it in the timing model. Then, it iterates through the current schedule up to the time of the new arrival and generates a list of completed operations, from the time of the previous arrival up to the last one: it removes each one from the timing model and updates the job's counters. Finally, if a job was completed, it fully removes it from the timing model.

5.3.2 Timing

The timing structure is based on a Gurobi model, that is update throughout the simulation and is responsible for producing a full schedule from the ordered queues generated by the agent.

It is composed by 4 main methods:

- *Initialize Timing method*: this method initializes the model with the first n_s static jobs and the corresponding constraints.
- *Insert Job method*: this function updates the model by adding the variables corresponding to each operation of the new job.

- *Delete Operation method*: this is called by the event manager each time an operation is completed and removes all the variables and constraints relative to it from the model. The method also checks if the operation was the last one of the corresponding job: if that is the case, the job's variables and constraints are removed from the model as well.

- *Timing Schedule method*: this method is responsible for producing the definitive schedule for the job shop. It takes the action of the agent as input and, after defining the objective function and adding the precedence constraints defined by the queues, solves the Gurobi model, minimizing the JIT penalty.

Chapter 6

Proposed method

In the proposed method, at each new job arrival a Time Simulation agent simulates time according to the current schedule and, each time a machine becomes idle and the operation for the new job on that machine, if existing, is feasible, refers to a Reinforcement Learning agent that decides whether to insert the new job's operation in the machine queue, previous to the one scheduled next.

The latter takes its decision based on the state features, i.e. the state representation of the machine's queue and the tardiness of the schedule, which are computed (or, if exact computation is not possible, estimated) by the Time Simulation agent.

6.1 Time Simulation Agent

The Time Simulation agent main functions are that of simulating the time by means of the current schedule and interact with the Reinforcement Learning agent, by providing to it all necessary information, like state features and estimated rewards, and inserting the new job's operation in the assigned position in the queue.

Besides the "get action" and "reset" methods, it is equipped with some auxiliary methods, necessary to calculate the features and rewards, to check the feasibility of actions and to restore the schedule once updated.

6.1.1 Main methods

The "Get action" method

This method is responsible for the simulation of the time and the interactions with the RL agent: in particular, it is composed of the following main steps.

- The agent reads the line corresponding to the new job and generates a graph of the job shop up to the new arrival time, necessary to check the feasibility of the RL agent's actions.
- For each operation of the new job, the agent adds a node to the graph, computes the time that the operations becomes feasible t_{end} (i.e. the time the previous operation is completed) and estimates the due date of such operation.
- If the queue of the corresponding machine is empty, the job is simply appended to it.
- Else, the agent starts iterating through the current schedule, while updating a copy of the operation counters.
- Each time the required machine becomes idle, if inserting the new operation in the corresponding queue index is a feasible action, it calls the RL agent to decide wheter to proceed with the insertion or not.
- If the RL agent decides to insert the new operation in the queue, the agent adapts the current schedule accordingly, setting the starting time of the operation in such a way that the penalty produced by the estimated due date is minimized. Then, the schedule feasibility is restored by checking that there are no overlapping operations.
- Finally, if the operation wasn't inserted up to the end of the machine's queue, the agent simply inserts it at the bottom of the queue.

The "reset" method

This method simply resets the Reinforcement Learning agent at each episode (i.e. each time a new operation is considered).

6.1.2 Auxiliary methods

The Time Simulation Agent was provided with some auxiliary methods, necessary to check the feasibility of the chosen actions and of the new schedule, to compute the state features to be fed to the agent and the estimated reward. The cited methods are the following:

- *"Check feasibility"*: this function checks if inserting the operation in the given index of the queue is feasible by inserting the corresponding edges in the graph and checking if the latter is still acyclic. If that's not the case, the method output is False and the corresponding action is not considered.
- *"Restore feasibility"*: this method iterates through the current schedule and restores feasibility in case of overlapping operations within the same machine's queue or the same job.
- *"Compute features"*: this function is responsible for computing the state features to be fed to the agent.
- *"Reward"*: this computes the estimate of the reward by means of the state features.
- *"Create graph"*: this is responsible for generating the initial graph containing all operations scheduled up to the last arrival time.

6.2 Reinforcement Learning Agent

Each time an insertion is feasible, the Time Simulation Agent refers to a Reinforcement Learning Agent to decide whether to proceed or not with the insertion: the latter was built using PyTorch and implements a Q-learning approach with Temporal Difference (TD) learning and function approximation.

In particular, the Q-learning values were approximated using a neural network, but other functions could be used, such as piece-wise linear functions or decision trees: this was necessary since the considered state features are continuous and therefore the number of states is infinite.

Another option would've been that of discretizing the state space, therefore adapting the problem to be solved via classical Q-learning tabular methods.

6.2.1 State Features

The state features are computed by the Time Simulation Agent each time an insertion is feasible and describe both the machine and the new job state, together with the global penalty of the current schedule.

The considered features are the following:

- *Job Shop Penalty*: this feature is computed by taking into account both the completely scheduled jobs and the new job's penalty. To estimate the latter I consider the estimation of the due date of the last inserted operation of the job.
- *New Job Penalty*: this simply corresponds to the penalty of the new job, based on the last inserted operation and the estimate of its due date.
- *Mean Tardiness*: this measures the mean tardiness of the jobs in the queue, following the considered index, without taking into consideration the tardiness weights.
- *Mean Earliness*: this measures the mean earliness of the jobs in the queue, following the considered index, without taking into consideration the earliness weights.

Note that the state features summarize all the information needed on the past states and decision, so that the state transition only depends on the present state of the job shop and the chosen action, therefore maintaining

the Markov Property and allowing us to solve the problem by means of Reinforcement Learning.

6.2.2 Action space

Each time the agent gets called, it decides whether to insert or not the new operation at the considered index of the queue: therefore, the action space is binary.

6.2.3 Temporal Difference Learning

Temporal Difference (TD) learning is a combination of Monte Carlo ideas and Dynamic Programming (DP) ideas. Like Monte Carlo methods,TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (by bootstrapping).

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of v_π for the non terminal states S_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. On the other hand, TD methods need to wait only until the next time step. At time t + 1 they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$.

The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + V(S_{t+1}) - V(S_t)\right] \tag{6.1}$$

The main difference is that the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + V(S_{t+1})$. This TD method is called TD(0), because it is a special case of the TD(λ) implemented in this work.

From Chapter 4, we know that

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s\right] \tag{6.2}$$

$$= \mathbb{E}_\pi \left[R_{t+1} + \lambda G_{t+1} \mid S_t = s\right]$$

$$= \mathbb{E}_\pi \left[R_{t+1} + \lambda v_\pi(S_{t+1}) \mid S_t = s\right] \tag{6.3}$$

Roughly speaking, Monte Carlo methods use an estimate of (6.2) as a target, whereas DP methods use an estimate of (6.3) as a target. In Monte

Carlo the target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (6.3) and it uses the current estimate V instead of the true v_π. Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP. (Sutton and Barto [2018])

The algorithm below specifies completely TD(0).

Tabular TD(0) for Estimating v_π

Input: Policy π to be evaluated
Algorithm parameter: Step size $\alpha \in (0, 1]$
Initialize $V(s)$ for all $s \in S$, arbitrarily except that $V(\text{terminal}) = 0$
for each episode **do**
    Initialize S
    while S is not terminal **do**
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha \left[R + V(S') - V(S)\right]$
        $S \leftarrow S'$
    end while
end for

TD predictions for control

To apply TD predictions for control, the first step is to learn an action-value function, $q_\pi(s, a)$, instead of a state-value function. This can be done either On-Policy or Off-Policy.

Sarsa: On-Policy TD

For an on-policy method, we estimate $q_\pi(s, a)$ for the current behavior policy π for all states s and actions a. This can be done using a TD method similar to the one used for learning v_π. The update rule for Q is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right], \qquad (6.4)$$

which is applied after every transition from a nonterminal state S_t. This rule uses all elements of the transition $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, leading to the name Sarsa for the algorithm.

In an on-policy control algorithm based on Sarsa, we continuously estimate q_π for the behavior policy π, while also adjusting π toward greediness with respect to q_π.

Q-learning: Off-Policy TD

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins and Dayan [1992]). Its simplest form, one-step Q-learning, is defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \quad (6.5)$$

In this case, the learned action-value function, Q, directly approximates q^*, the optimal action-value function, independent of the policy being followed. This greatly simplifies the analysis of the algorithm and enabled early convergence proofs.

Eligibility Traces and TD(λ)

Eligibility traces are one of the basic mechanisms of reinforcement learning: almost any temporal-difference (TD) method, such as Q-learning or Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more eciently. Eligibility traces unify and generalize TD and Monte Carlo methods. When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end ($\lambda = 1$) and one-step TD methods at the other ($\lambda = 1$). In between are intermediate methods that are often better than either extreme method. Eligibility traces also provide a way of implementing Monte Carlo methods online and on continuing problems without episodes.

In this work, we define $TD(\lambda)$ mechanistically, instead of presenting its forward view. The mechanistic, or backward, view of $TD(\lambda)$ is useful because it is simple conceptually and computationally and provides a causal, incremental mechanism for approximating the forward view and, in the off-line case, for achieving it exactly. In fact, the forward view itself is not directly implementable because it is acausal, using at each step knowledge of what will happen many steps later.

In the backward view of $TD(\lambda)$, each state s has an associated eligibility trace, denoted $E_{t(s)} \in \mathbb{R}^+$, which is a memory variable. The eligibility trace for a non-visited state $s \neq S_t$ at time t decays by the factor $\gamma\lambda$ on each step:

$$E_t(s) = \gamma\lambda E_{t-1}(s), \quad \forall s \in S,\, s \neq S_t, \tag{6.6}$$

where γ is the discount rate and λ is the trace-decay parameter.

For the visited state S_t, the eligibility trace decays similarly but is incremented by 1:

$$E_t(S_t) = \gamma\lambda E_{t-1}(S_t) + 1. \tag{6.7}$$

This type of eligibility trace is known as an accumulating trace, as it builds up each time a state is visited and gradually fades away when it is not visited.

Eligibility traces indicate the degree to which each state is eligible for learning updates if a reinforcing event occurs. The reinforcing events of interest are the moment-by-moment one-step TD errors. For example, the TD error for state-value prediction is given by:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t). \tag{6.8}$$

In the backward view of $TD(\lambda)$, the global TD error signal triggers updates proportional to the TD error for all recently visited states, based on their eligibility traces:

$$\Delta V_t(s) = \alpha\delta_t E_t(s), \quad \text{for all } s \in S. \tag{6.9}$$

These updates can be applied step-by-step to form an on-line algorithm or deferred until the end of an episode to create an off-line algorithm.

A complete algorithm for online $TD(\lambda)$ is defined below.

Online tabular $TD(\lambda)$

Initialize $V(s)$ arbitrarily
for each episode **do**
    Initialize $E(s) = 0$ for all $s \in S$
    Initialize state S
    while S is not terminal **do**
        $A \leftarrow$ action given by policy π for state S
        Take action A, observe reward R, and next state S'
        $\delta \leftarrow R + \gamma V(S') - V(S)$
        Update eligibility trace for S
        $E(S) \leftarrow E(S) + 1$ ▷ (accumulating traces)
        for all $s \in S$ **do**
            $V(s) \leftarrow V(s) + \alpha\delta E(s)$
            $E(s) \leftarrow \gamma\lambda E(s)$
        end for
        $S \leftarrow S'$
    end while
end for

Sarsa(λ)

How can eligibility traces be used not just for prediction, as in $TD(\lambda)$, but for control? As usual, one popular approach is to learn action values, $Q_t(s, a)$, rather than state values, $V_t(s)$.

The idea in $Sarsa(\lambda)$ is to apply the $TD(\lambda)$ prediction method to state–action pairs instead of just states. Therefore, we need a trace for each state–action pair. Let $E_t(s, a)$ denote the trace for state–action pair (s, a).

The traces are updated in the same way as before, except they are triggered by visiting the state–action pair, and the update rule becomes:

$$E_t(s, a) = \gamma\lambda E_{t-1}(s, a) + I\{S_t = s\}I\{A_t = a\}, \quad \forall s \in S,\, a \in A, \qquad (6.10)$$

where $I\{\cdot\}$ is the identity-indicator function. Apart from this, $Sarsa(\lambda)$ is just like $TD(\lambda)$, except that state–action variables are used instead of state variables:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha\delta_t E_t(s, a), \quad \forall s, a, \qquad (6.11)$$

where the TD error δ_t is given by:

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t). \tag{6.12}$$

$\mathbf{Q}(\lambda)$

When Watkins and Dayan [1992] introduced Q-learning, they also proposed an approach to combine it with eligibility traces. However, when using Q-learning with eligibility traces special care is required because learning about the greedy policy is valid only as long as the greedy policy is followed.

Thus, in $Q(\lambda)$, if an exploratory action is taken, the lookahead stops, unlike $TD(\lambda)$ and $Sarsa(\lambda)$, which continue to look ahead until the end of the episode. The eligibility traces are updated similarly to $Sarsa(\lambda)$, but with one key difference: they are reset to zero whenever an exploratory action is taken.

The trace update occurs in two steps: first, the traces are decayed by $\gamma\lambda$, or set to zero if an exploratory action was taken. Second, the trace for the current state–action pair is incremented by 1. The overall result is:

$$E_t(s,a) = \begin{cases} \gamma\lambda E_{t-1}(s,a) + I_{\{S_t=s\}} \cdot I_{\{A_t=a\}}, & \text{if } Q_{t-1}(S_t, A_t) = \max_a Q_{t-1}(S_t, a), \\ I_{\{S_t=s\}} \cdot I_{\{A_t=a\}}, & \text{otherwise.} \end{cases} \tag{6.13}$$

The rest of the algorithm is defined by:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t E_t(s,a), \quad \forall s \in S,\, a \in A(s), \tag{6.14}$$

where the TD error δ_t is:

$$\delta_t = R_{t+1} + \gamma \max_{a'} Q_t(S_{t+1}, a') - Q_t(S_t, A_t).$$

Clearly, cutting off traces whenever an exploratory action is taken diminishes the benefit of using eligibility traces. If exploratory actions are frequent, as is often the case early in learning, then multi-step backups will rarely occur, and learning may not be much faster than one-step Q-learning.

6.2.4 Function Approximation

Reinforcement learning systems must generalize effectively to be applicable in artificial intelligence and large-scale engineering applications, especially when either the state space or the action space are not finite.

This can be accomplished by employing various supervised-learning function approximation methods, such as artificial neural networks, decision trees, and multivariate regression techniques.

However, not all function approximation methods are equally suitable for reinforcement learning. Many advanced neural network and statistical methods assume a static training set, which is processed through multiple passes.

In reinforcement learning, it is crucial for learning to occur online, while interacting with the environment or its model. This requires methods that can efficiently learn from incrementally acquired data. Additionally, reinforcement learning often needs function approximation methods that can handle nonstationary target functions, which change over time.

To evaluate function approximation methods, appropriate performance measures are necessary. Most supervised learning methods aim to minimize the root-mean-squared error (RMSE) over a distribution of inputs. In the context of value prediction, the inputs are states, and the target function is the true value function v_π. The RMSE for an approximation $\hat{v}$, using parameter w, is given by:

$$\text{RMSE}(w) = \sqrt{\sum_{s \in S} d(s) \left[v_\pi(s) - \hat{v}(s, w)\right]^2}, \tag{9.1}$$

where $d : S \to [0, 1]$ represents a distribution over the states such that $\sum_s d(s) = 1$, indicating the relative importance of errors in different states.

Gradient-descent methods offer a natural extension for function approximation, including techniques developed with eligibility traces. Linear gradient-descent methods are theoretically appealing and perform well with appropriate features. Choosing these features is a crucial way to incorporate prior domain knowledge into reinforcement learning systems.

Examples of linear methods include radial basis functions, tile coding, and Kanerva coding.Nonlinear gradient-descent function approximation methods include backpropagation for multilayer neural networks.

6.2.5 Proposed architecture

In the proposed approach, the Reinforcement Learning agent learns via online Temporal Difference learning with eligibility traces: more in particular, the agent was implemented in such a way that it can learn either using $Sarsa(\lambda)$ or $Q(\lambda)$.

To approximate the Q-value function, a neural network was used and trained using gradient descent. The gradient descent equation for neural network parameters is given by

$$\theta \leftarrow \theta + \alpha \cdot \nabla J(\theta)$$

Then, the TD(λ) state-value update equation becomes

$$Q(s,a) \leftarrow Q(s,a) + \alpha[Q_t - \hat{Q}(S_t, A_t)]E_t(s,a)$$

The objective is to improve the estimate of $Q(s,a)$ by minimizing the difference between the target and predicted action-values. The mean squared error between these two values was used as the loss function $J(\theta)$.

$$J(\theta) = \frac{1}{2} \left\| [Q_t - Q(S_t, A_t; \theta)] \cdot E_t(s,a) \right\|^2$$

The gradient descent equation for TD(λ) is therefore

$$\theta \leftarrow \theta + \alpha \cdot \delta_t \cdot E_t(s,a) \cdot \nabla_\theta Q(S_t, A_t; \theta)$$

where

$$\delta_t = Q_t - Q(S_t, A_t; \theta)$$

To be specific, two different neural networks were used: while Q_t refers to the target output obtained at time step t, we bear in mind that the target output comprises the immediate reward plus the discounted estimate of future returns obtainable from the next state until the agent reaches the goal. That estimate is also a prediction, just like $Q(S_t, A_t; \theta)$. Technically, Q_t and $Q(S_t, A_t; \theta)$ could be obtained from the same network. However, since the network weights are updated at every step, as $Q(S_t, A_t; \theta)$ improves, Q_t will change (not necessarily improve), and the method will be chasing a moving target with no end.

Having two neural networks allows to decouple $Q(S_t, A_t; \theta)$ and Q_t to some extent for the calculation of the mean squared error, so that the training becomes more stable.

Thus, I defined:

- Q_{main}, the main network that we will train to get $Q(S_t, A_t; \theta)$. As training occurs at every step, we can expect fluctuations here.
- Q_{target}, the secondary network which we will use to obtain Q_t. This network is not trained but rather, updated periodically (via a soft update) with the learned weights from Q_{main} to help it improve with fewer fluctuations than Q_{main}.

A memory buffer is used to collect the outputs (current state and action, reward obtained, next state and action) of each step, from which random baches are sampled to train the neural network Q_{main}.

This batch update method is preferred because it is generally more stable than single-sample updates (the latter has high variance and may lead to slow or non-convergence).

This is also where we compute Q_t, where:

- If the agent is acting *on-policy*, then $Q(S_{t+1}, A_{t+1})$ will be obtained from the Q_{target} network using the next action taken (by policy), i.e.,

$$Q_t = [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})]$$

- If the agent is *not acting on-policy*, then $Q(S_{t+1}, a)$ will be obtained from the Q_{target} network using the action that gives the best Q-value, i.e.,

$$Q_t = \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a)\right]$$

Chapter 7

Numerical Results

In this section we focus on the numerical results obtained by applying the proposed approach.

We start by introducing the settings of our instance generator, i.e. on the features of the considered job shop:

Table 7.1: Instance Generation Settings

Parameter	**Value**
Due date type	Loose
Ealiness-Tardiness Trade Off	Equal
First arrival time	200
Interarrival time distribution	Exponential
Average arrival time	75
Average operations per jobs	7
Initial number of jobs	10

These are the main characteristics that all instances, either used in training or testing, share. Then, the number of dynamic jobs was increased in the training phase, to reduce the number of environment's restarts and present the agent with a wide range of states, while it was kept lower in the test instances.

Two different agents were trained, implementing $Q(\lambda)$ and $Sarsa(\lambda)$. Then, the performances of both agents were tested on various instances and compared with that of a FIFO agent, simply inserting all the operations belonging to the new job at the end of the corresponding queues.

7.1 Training

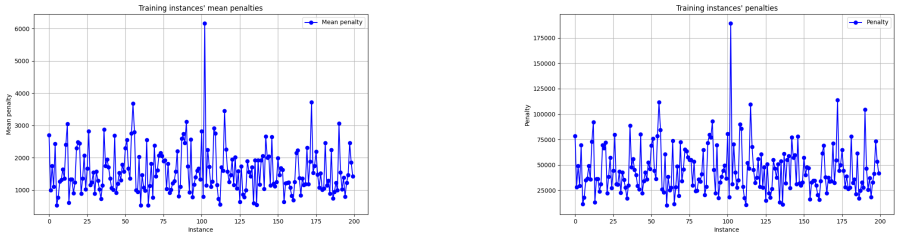
Training of each agent was carried out on 200 instances, each containing $n_s = 10$ static jobs and $n_d = 100$ dynamic jobs. Since each job contains on average 7 tasks, so for each instance, the agent interacts with more than 500 episodes for instance (where an episode is considered as the insertion of a single task).

The table below contains the chosen values of the method's hyperparameters:

Table 7.2: Training Settings

Hyperparameter	**Value**
Decaying Rate λ	0.99
Learning Rate α	0.05
Discount Factor γ	0.9
Exploration rate ϵ	0.1
Memory buffer size	10000
Batch Size	32

First, the SARSA agent (the on-policy agent) was trained: to evaluate the agent's performance during training, it was decided to plot the weighted average penalty per job, calculated as the sum of the weighted tardiness and earliness and the weighted penalty related to the flowtime.



(a) Mean penalty per instance

(b) Penalty per instance

Figure 7.1: Training performances - Sarsa agent

However, as we can observe, these results are particularly affected by variance, taking values over a wide range, and at first glance, they do not seem to show any improvement in the agent's performance (7.1a). The same observations can be made about the final penalty for each instance (7.1b).

However, by plotting the simple moving averages of these two measures, calculated over the previous 20 instances, a clear improvement can be observed, with averages that initially reach values close to 2000 and after the first 100 instances stabilize below 1600 for the first measure (7.2), and that go up to 56000 and then stabilize below 44000 for the second (7.3).

Figure 7.2: SMA of mean penalty per instance - SARSA agent

Similar conclusions were drawn on the training of the Off Policy Agent (the Q-learning agent): for this reason, only the plots of the Simple Moving Averages of the performances are shown here. As we can observe, the agent apparently seems to perform better than the previous one: the SMA of the mean penalty stabilizes under 1500 after the first 100 instances, while the SMA of the final penalty reaches values under 40000 (7.4, 7.5).

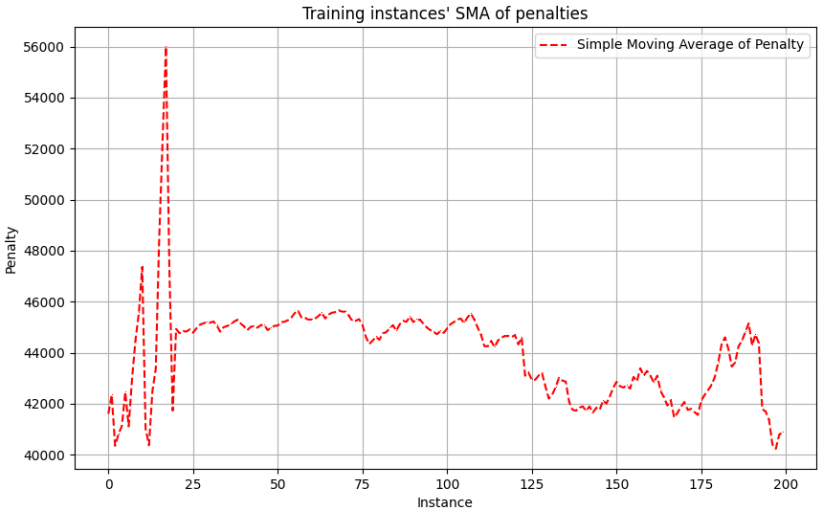


Figure 7.3: SMA of penalty per instance - SARSA agent

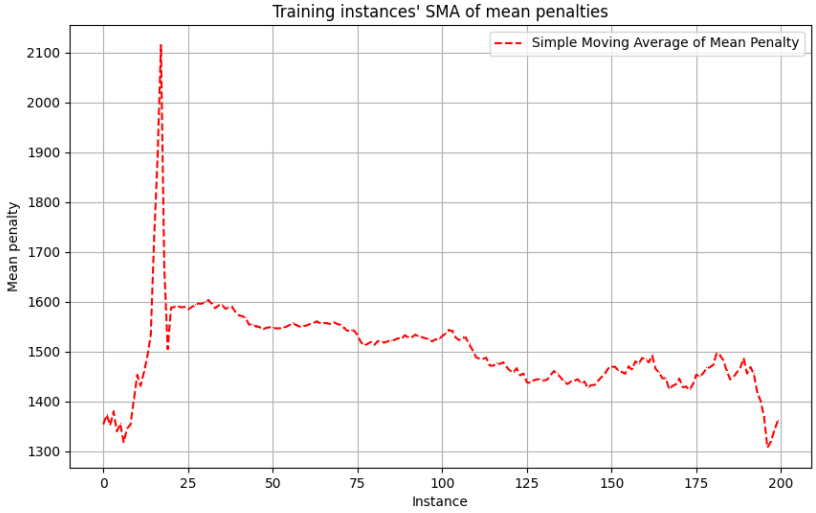


Figure 7.4: SMA of mean penalty per instance - Q-learning agent

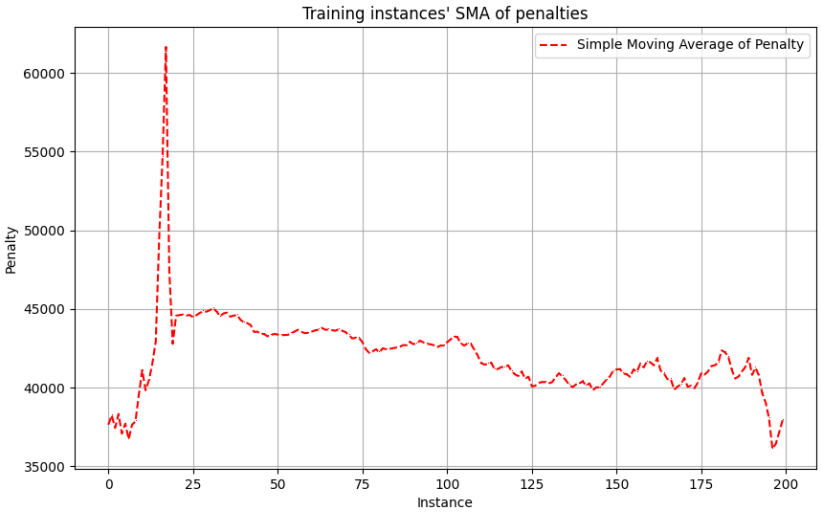


Figure 7.5: SMA of penalty per instance - Q-learning agent

7.2 Evaluation

Our approach was tested on a wide range of instances, containing from 10 up to 50 dynamic jobs. The results were compared with those obtained by using a FIFO agent, proving that the proposed method is able to outperform it on almost all instances.

The FIFO agent, built to compare its results with those of the proposed method, operates in the following simple way: each time a new job arrives, it inserts each operation at the end of the queue of the corresponding machine.

The agents were tested on 3 groups of instances, each composed of 100 instances containing 10,20 and 50 dynamic jobs respectively.

The FIFO agent was able to outperform the built agents in a small percentage of instances when working with 10 or 20 dynamic jobs. However, once the dimension of the problem increased, the FIFO agent was outperformed on all instances.

At first, the two agents were tested on a group of small instances, containing 10 dynamic jobs. The mean final penalties obtained by the three agents are shown in the table below.

Table 7.3: Test results - 10x10 instances

Agent	**Mean**
SARSA	1620.22
Q-learning	1679.93
FIFO	1805.38

As we can notice, the mean performance of the two agents seems to be much superior than that of the FIFO agent. However, when observing the winning count of the three agents (i.e. the percentage of instances on which each agent was able to outperform the remaining two), we can observe how the FIFO agent was able to surmount the other two in 18% of the instances (7.6).

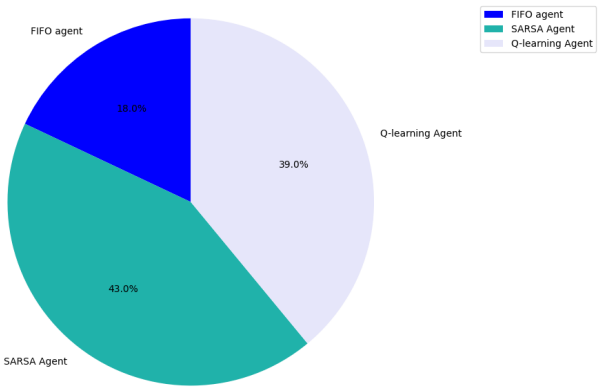


Figure 7.6: Winning count % per agent - 10x10 instances

More particularly, the SARSA agents seems to perform better than the Q-learning one, obtaining the lowest penalty on 43% of the instances. This is also confirmed by the graphs below, that show the penalty difference between those obtained by the FIFO agent and the tested one:

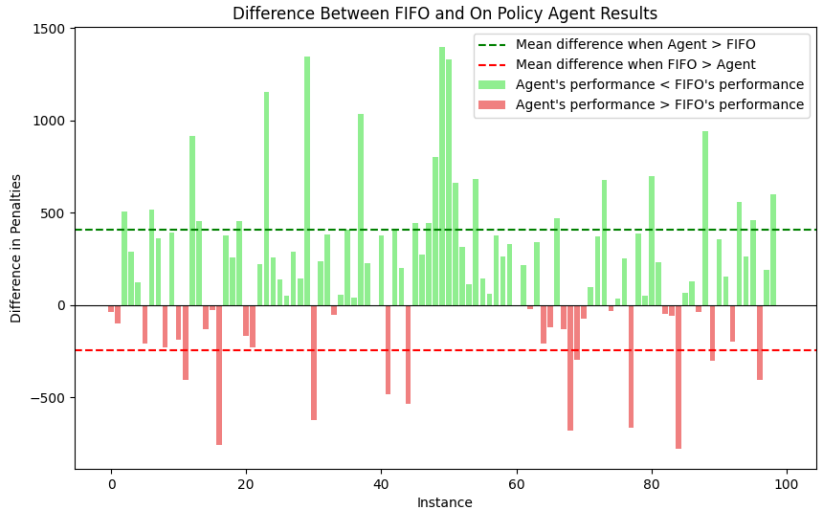


Figure 7.7: SARSA agent penalty differences - 10x10 instances

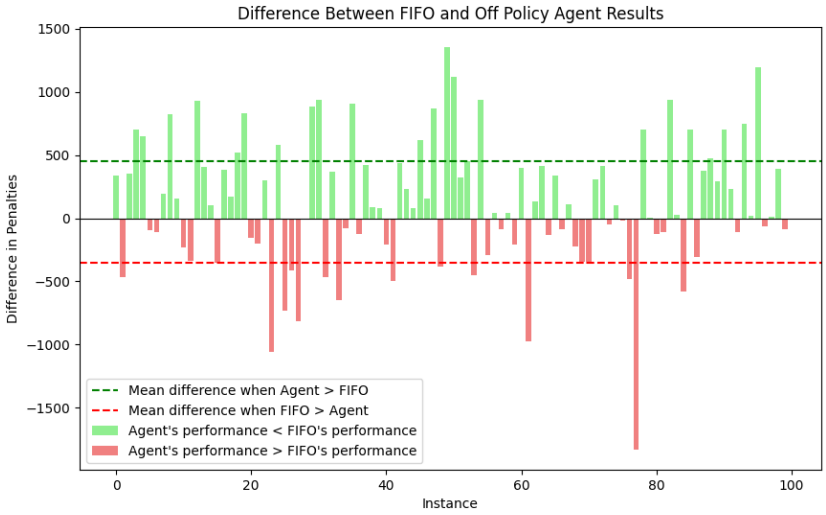


Figure 7.8: Q-learning agent penalty differences - 10x10 instances

The SARSA agent clearly outperforms the FIFO agent on an higher number of instances than the Q-learning agent and, whenever the agent is

outperformed by the FIFO agent, the mean difference is equal to 243.35 (7.7), compared to a mean of 349.17 for the Q-learning agent.

However, when outperforming the FIFO agent, the Q-learning agent reaches an higher mean difference than the SARSA one, as we can observe (7.8).

When increasing the number of dynamic jobs, the percentage of instances where the FIFO agent performance excedeed that of the other two agents decreased: when tested on instances with 20 dynamic jobs, the two proposed agents outperformed the FIFO one on 95% of them (7.9).

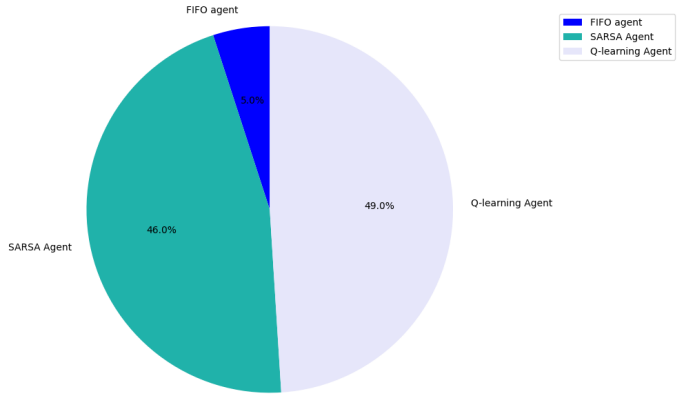


Figure 7.9: Winning count % per agent - 10x20 instances

The difference in the mean penalty per instance increased as well, as shown in the table below:

Table 7.4: Test results - 10x20 instances

Agent	**Mean**
SARSA	4829.29
Q-learning	4716.01
FIFO	5911.04

On the other hand, as we can note from the results shown above, the Off-policy agent started to outperform the On-policy agent, obtaining the best performance on 49% of instances.

However, as before, the results obtained with the Q-learning agent are less stable than those obtained with the SARSA agent: when outperforming the FIFO agent, the mean performance difference is higher; however, the same conclusions can be drawn for the case in which the FIFO agent outperforms the tested one (7.10,7.11).

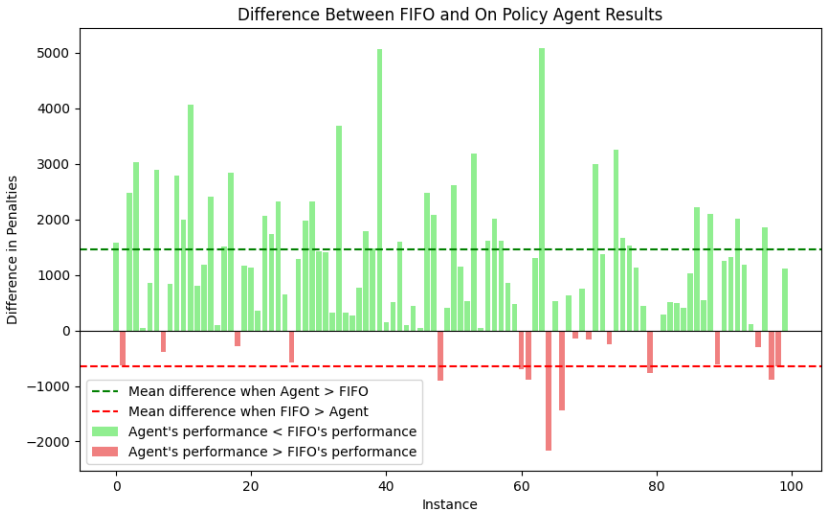


Figure 7.10: SARSA agent penalty differences - 10x20 instances

Moreover, when the SARSA agent is outperformed, the difference is lower than the mean difference for most instances. As for the Q-learning agent, this happens on only 50% of the instances on which the agent was outmatched.

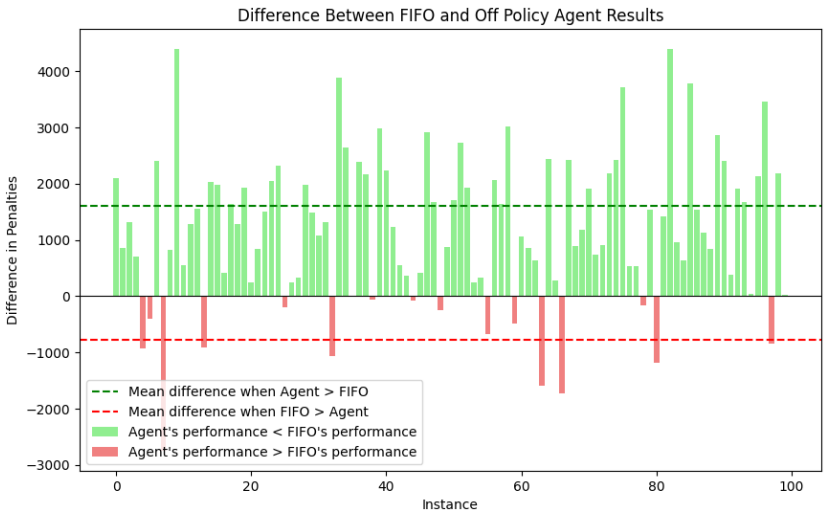


Figure 7.11: Q-learning agent penalty differences - 10x20 instances

Finally, when tested on instances with 50 dynamic jobs, the two agents were able to fully outperform the FIFO agent performances 7.12.

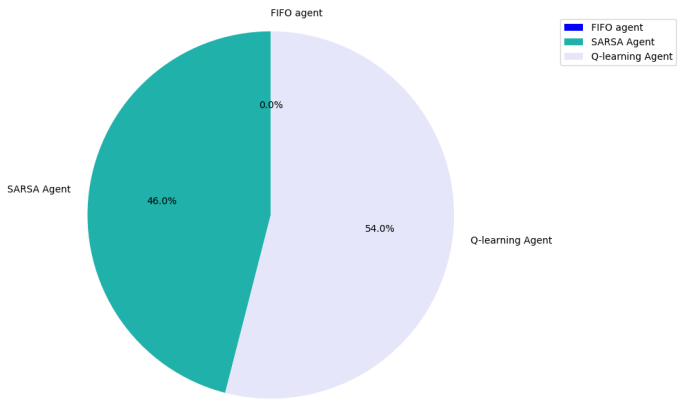


Figure 7.12: Winning count % per agent - 10x50 instances

As above, the Q-learning agent reached the best performance on an higher

percentage of instances, compared to the SARSA agent. Moreover, as shown in the table below, the mean penalty obtained by the Q-learning agent results lower than that obtained by the SARSA one.

Table 7.5: Test results - 10x50 instances

Agent	**Mean**
SARSA	25315.87
Q-learning	24877.11
FIFO	34754.28

On the other hand the results seem to be more stable when using SARSA, confirming what seen before: when outperformed by the FIFO agent, the Q-learning agent shows a mean difference in perfomances much higher than the SARSA agent and the obtained difference on the single instance is higher than such mean in 2 instances out of 3 (7.13,7.14).

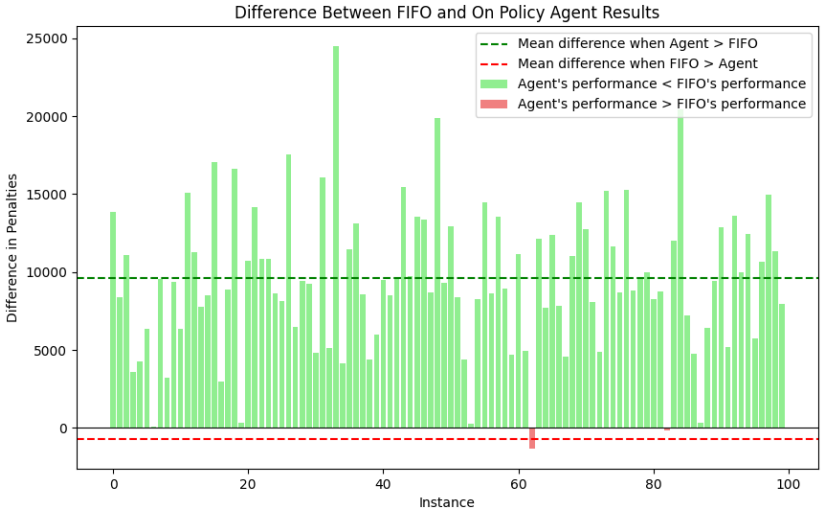


Figure 7.13: SARSA agent penalty differences - 10x50 instances

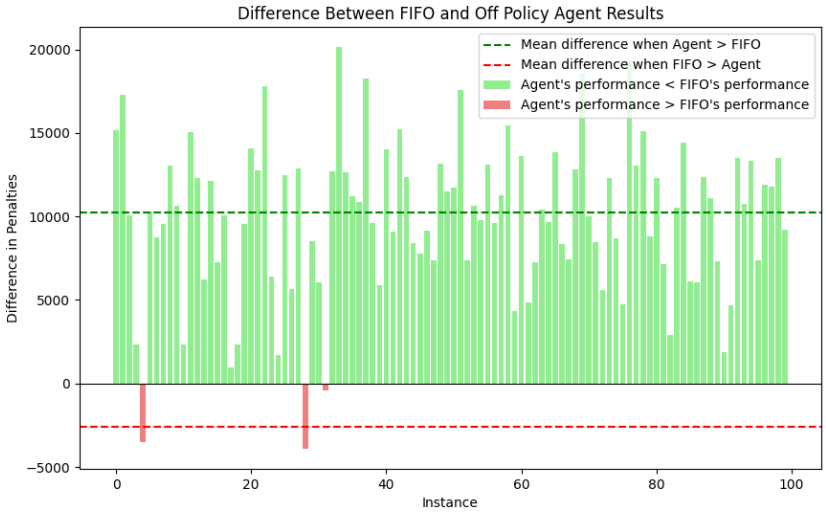


Figure 7.14: Q-learning agent penalty differences - 10x50 instances

For both agents, the performances result to be way higher than those of the FIFO agent, outperforming it in respectively 98% and 97% of the instances and confirming the superiority of the proposed method: in both cases the mean difference in penalty when outperforming the FIFO agent is in a neighborhood of 10000 units.

Chapter 8

Conclusions

This thesis contributes to the research on Job Shop Scheduling with Dynamical Arrivals and more in general to the field of Operational Research. The primary contributions are twofold: both in the areas of implementation and modeling, and in the development of a benchmark solution, obtained with a "multiple operation" point of view, which was rarely explored before.

First of all, this project lays the foundation for an extensive research on the topic of DJSP, having defined and implemented a custom simulation environment, which can be adapted to both 'single operation' and 'multiple operation' approaches, as well as to a wide range of agents for each. This was achieved through an extensive literature review on job shop scheduling problems under uncertainty and the study of the best possible structures for modeling the problem.

Secondly, this work focuses on the development of a new method for periodic predictive-reactive scheduling, based on a reinforcement learning agent trained using temporal differences, and comparing it with a simple heuristic: specifically, two agents were implemented, a SARSA agent and a Q-learning agent. These interacted with all machines in the job shop, observing their state features and choosing how to insert the new job's operations in their queues. For both, function approximation was used to adapt the method to the continuous state space.

As shown in chapter 7, the two agents outperform the FIFO agent in most cases, completely surpassing it when the number of dynamic jobs reaches 50. The performances of the FIFO agent outperform those of the proposed method on a low percentage of the smaller instances, suggesting that the

method is better suited to outperform simple heuristics and dispatching rules in large-scale problems.

Moreover, the training was conducted only on 200 instances for each agent, due to the computational costs, but the training performances' graphs suggest that a more exhaustive training could lead to better results.

Finally, an interesting comparison would be that of the obtained results with the results that could be obtained if the problem was static and all $n_s + n_d$ jobs were known at time 0, using state-of-art methods for the Static Job Shop Problem.

This work leaves room for further research and generalizations: for example, structural changes could be made, such as the choice of the RL agent type or the Q-value approximation function. On the other hand, the method could be generalized to instances with stochastic processing times, developing an agent for each individual machine, rather than a single agent for the entire job floor.

Finally, the built environment can be generalized to consider all types of dynamic disruptions, such as machine breakdowns, cancellation of orders, etc., thus paving the way for broad research on the topic of dynamical job shops.

Bibliography

J. M. Ana Esteso, David Peidro and M. Díaz-Madroñero. Reinforcement learning applied to production planning and control. *International Journal of Production Research*, 2023.

A. Baykasoğlu and F. S. Karaslan. Solving comprehensive dynamic job shop scheduling problem by using a grasp-based approach. *International Journal of Production Research*, 2017.

N. Kundakcı and O. Kulak. Hybrid genetic algorithms for minimizing makespan in dynamic job shop scheduling problem. *Computers & Industrial Engineering*, 2016.

S. R. Lawrence and E. C. Sewell. Heuristic, optimal, static, and dynamic schedules when processing times are uncertain. *Journal of Operations Management*, 1997.

R. Liu, R. Piplani, and C. Toro. A deep multi-agent reinforcement learning approach to solve dynamic job shop scheduling problem. *Computers & Operations Research*, 2023.

M. Morady Gohareh and E. Mansouri. A simulation-optimization framework for generating dynamic dispatching rules for stochastic job shop with earliness and tardiness penalties. *Computers & Operations Research*, 2022.

M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., USA, 1st edition, 1994.

C. Rajendran and O. Holthaus. A comparative study of dispatching rules in dynamic flowshops and jobshops. *European Journal of Operational Research*, 1999.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Z. Wang, J. Zhang, and S. Yang. An improved particle swarm optimization algorithm for dynamic job shop scheduling problems with random job arrivals. *Swarm and Evolutionary Computation*, 2019.

C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 1992.

X. Wu and X. Yan. A spatial pyramid pooling-based deep reinforcement learning model for dynamic job-shop scheduling problem. *Computers & Operations Research*, 2023.

T. Zhang, S. Xie, and O. Rose. Real-time job shop scheduling based on simulation and markov decision processes. In *2017 Winter Simulation Conference (WSC)*, 2017.

List of Tables

List of Figures

Acknowledgments

First of all, I would like to thank Professors Fadda and Brandimarte for guiding me throughout this journey, and Lorenzo and Valerio for their essential help in completing this project.

I would then like to extend a heartfelt thanks to my family, who has always encouraged me to pursue every dream I had and nurture my passion in mathematics.