TU WIEN Informatics

Graph-Based Multi-Agent Reinforcement Learning im Dynamischen und Flexiblen Job-Shop Scheduling Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Daniel Mazanek, B.Sc.
Matrikelnummer 12005060

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Clemens Heitzinger
Mitwirkung: Univ.Ass. Dipl.-Ing. Lorenz Florian Fink, BSc
Assistant Prof. Dipl.-Ing. Dr.techn. Thomas Fabian Trautner

Wien, 2. Juli 2026

Daniel Mazanek	Clemens Heitzinger

TU WIEN Informatics

Graph-Based Multi-Agent Reinforcement Learning on the Dynamic and Flexible Job-Shop Scheduling Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Daniel Mazanek, B.Sc.
Registration Number 12005060

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Clemens Heitzinger
Assistance: Univ.Ass. Dipl.-Ing. Lorenz Florian Fink, BSc
Assistant Prof. Dipl.-Ing. Dr.techn. Thomas Fabian Trautner

Vienna, July 2, 2026

Daniel Mazanek | Clemens Heitzinger

Erklärung zur Verfassung der Arbeit

Daniel Mazanek, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 2. Juli 2026

Daniel Mazanek

Danksagung

Ich möchte mich bei meinem Betreuer, Prof. Dr. Clemens Heitzinger, für die Betreuung dieser Arbeit und die Unterstützung während des gesamten Arbeitsprozesses bedanken.

Außerdem möchte ich mich bei Univ.Ass. Dipl.-Ing. Lorenz Florian Fink, BSc, und Assistant Prof. Dipl.-Ing. Dr.techn. Thomas Fabian Trautner für die intensive Unterstützung, das Feedback und die vielen hilfreichen Diskussionen während der Entwicklung dieser Arbeit bedanken.

Weiters möchte ich mich bei meiner Familie und meinen Freunden für ihre kontinuierliche Unterstützung und Motivation während meines Studiums bedanken. Ihre Geduld und ihr Zuspruch haben mir besonders in herausfordernden Phasen dieser Arbeit geholfen, vor allem während jener sieben Monate, in denen ich ihnen regelmäßig gesagt habe, dass ich in zwei Wochen fertig sein werde.

Ein besonderer Dank gilt Dominik Pichler, der in seiner Masterarbeit die GNNs analysiert und implementiert hat. Die Zeit, die wir mit Diskussionen über Implementierungsdetails und theoretische Aspekte verbracht haben, hätte vermutlich als zusätzliche ECTS in Kommunikationsfähigkeit angerechnet werden können.

Abschließend möchte ich mich bei allen bedanken, die direkt oder indirekt zur Fertigstellung dieser Arbeit beigetragen haben.

Acknowledgements

I would like to thank my supervisor, Prof. Dr. Clemens Heitzinger, for supervising this thesis and for providing guidance throughout the work.

I would also like to thank Univ.Ass. Dipl.-Ing. Lorenz Florian Fink, BSc, and Assistant Prof. Dipl.-Ing. Dr.techn. Thomas Fabian Trautner for their intensive support, feedback, and the many helpful discussions during the development of this thesis.

Furthermore, I would like to thank my family and friends for their continuous support and encouragement throughout my studies. Their patience and motivation helped me during challenging phases of this work, especially during the seven months in which I repeatedly told them I would be finished in two weeks.

A special thanks goes to Dominik Pichler, who analyzed and implemented the GNNs in his master's thesis. The amount of time we spent discussing implementation details and theoretical aspects could probably have been credited as additional ECTS in communication skills.

Finally, I would like to thank everyone who contributed directly or indirectly to the completion of this thesis.

Kurzfassung

Effizientes Scheduling ist eine zentrale Anforderung moderner Smart-Manufacturing-Umgebungen. Produktionsumgebungen sind zunehmend durch wechselnde Anforderungen, flexible Maschinenfähigkeiten und stochastische Ereignisse wie Verzögerungen oder maschinenbezogene Störungen geprägt. Das dynamische und flexible Job-Shop Scheduling Problem (JSSP) modelliert diese Herausforderungen.

Diese Arbeit untersucht Graph-basiertes Multi-Agent Reinforcement Learning (MARL) für das dynamische und flexible JSSP. Der Fokus der Arbeit ist zweigeteilt. Erstens wird analysiert, wie sich die architektonischen Hyperparameter und die Regularisierung des von den Agenten verwendeten neuronalen Netzwerks auf die Konvergenz im flexiblen JSSP auswirken. Dafür werden Multi-Layer Perceptron (MLP)- und Convolutional Neural Network (CNN)-Architekturen über verschiedene Instanzgrößen hinweg verglichen. Zweitens wird analysiert, welche dynamischen Faktoren von einem Graph-basierten MARL-Scheduling-Algorithmus gelernt werden können. Die zwei getesteten dynamischen Faktoren sind Verzögerungen, die durch Maschinen verursacht werden, und Verzögerungen, die durch Jobs verursacht werden.

Jobs und Maschinen werden durch eine Graph-basierte Repräsentation dargestellt. Die daraus entstehenden Embeddings werden verwendet, um Jobs für Maschinen auszuwählen. Die Performanz der Modelle wird anhand von Makespan und Utilization bewertet und mit einem statischen Modell sowie einfachen regelbasierten Baselines verglichen.

Die Ergebnisse zeigen, dass die getesteten MLP-Architekturen besser geeignet waren als die getesteten CNN-Architekturen. Eine Erhöhung der Anzahl der Schichten verbesserte die Ergebnisse nicht. Dies deutet darauf hin, dass die Architektur des neuronalen Netzwerks zur gewählten Zustandsrepräsentation passen muss.

Für die dynamischen Experimente zeigen die Ergebnisse einen deutlichen Unterschied zwischen Maschinenverzögerungen und Jobverzögerungen. Maschinenverzögerungen erzeugen ein konsistentes Lernbarkeitssignal. Die dynamische Policy verbessert sich in allen Maschinenverzögerungs-Szenarien gegenüber der statischen Policy und erreicht in den meisten Konfigurationen den niedrigsten Makespan. Jobverzögerungen zeigen ebenfalls eine konsistente, aber deutlich schwächere Verbesserung gegenüber der statischen Policy. Diese Verbesserung ist jedoch klein, und die dynamische Policy dominiert die heuristischen Baselines nicht in allen Konfigurationen. Die Ergebnisse deuten daher darauf hin, dass dynamische Faktoren nicht gleichermaßen lernbar sind.

Abstract

Efficient Scheduling is an essential requirement for modern Smart Manufacturing. Production environments are increasingly characterized by changing requirements, flexible machine capabilities, and stochastic events such as delays or machine-related disturbances. The dynamic and flexible JSSP models these challenges, but its combinatorial search space makes exact solutions infeasible for larger instances.

This thesis investigates Graph-based MARL on the dynamic and flexible JSSP. The focus of the work is twofold. First, the thesis analyzes how the architectural hyperparameters and regularization of the neural network used by the agents affect convergence on the flexible JSSP. For this purpose, MLP and CNN architectures are compared across different instance sizes. Second, the thesis analyzes which dynamic factors can be learned by a Graph-based MARL Scheduling algorithm. The two dynamic factors tested are delays caused by machines and delays caused by jobs.

To answer these questions, a Shopfloor environment was implemented that can generate static, flexible, and dynamic JSSP instances. Jobs and machines are represented through a Graph-based state representation. The resulting embeddings are used by the agents to select jobs for machines. The performance of the trained models is evaluated using makespan and utilization and is compared against a static model and simple rule-based baselines.

The results show that the tested MLP architectures were more suitable than the tested CNN architectures. The best results were achieved with a comparatively small MLP using two layers and dropout of 0.1. Increasing the number of layers did not improve the results, and CNN architectures did not converge well in the tested setup. This indicates that the neural network architecture must fit the chosen state representation.

For the dynamic experiments, the results show a clear difference between machine delays and job delays. Machine delays produced a strong and consistent learnability signal. The dynamic policy improved over the static policy across all machine-delay scenario medians and achieved the best makespan in most machine-delay configurations. Job delays also showed a consistent but much weaker improvement over the static policy. However, this improvement was small and the dynamic policy did not dominate the heuristic baselines in all configurations. This suggests that dynamic factors are not equally learnable. Learnability depends on where the disturbance is located in the Shopfloor and how actionable the disturbance pattern is for the scheduling policy.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation and Problem Statement 1
1.2 Research Question and Research Objective 4
1.3 Methodology 4
1.4 Scientific Relevance 6

2 Fundamentals **7**
2.1 Smart Manufacturing and Industry 4.0 7
2.2 Scheduling Problems 13
2.3 The Job-Shop Scheduling Problem 16
2.4 Reinforcement Learning 19
2.5 Graphs 34
2.6 Mapping the Requirements of Smart Manufacturing with MARL and a Graph-Based Representation 37

3 Related Work **41**
3.1 Works Identified 41
3.2 Commonalities and Differences of Identified Works 43
3.3 Research Gaps Addressed by This Thesis 46
3.4 Different Approaches Not Covered in This Thesis 47

4 Proposed Architecture **49**
4.1 Shopfloor Architecture 49
4.2 Algorithm 62

5 Experimental Setup **77**
5.1 Experimental Pipeline 77
5.2 Instance Generation Protocol 78

5.3 Learning Setup . . . 80
5.4 Experimental Design for RQ1 . . . 82
5.5 Experimental Design for RQ2 . . . 83
5.6 Evaluation . . . 85

6 Results and Interpretations **87**
6.1 RQ1 . . . 87
6.2 RQ2 . . . 90

7 Discussion **103**
7.1 Summary of the Main Findings . . . 103
7.2 Analysis of RQ1: Architecture and Representation . . . 104
7.3 Analysis of RQ2: Learnability of Dynamic Factors . . . 105
7.4 Methodological Implications of Order Generation on the JSSP . . . 107
7.5 Offline vs. online learning . . . 109
7.6 Limitations and Threats to Validity . . . 110
7.7 Future Work . . . 112

8 Conclusion **115**

A Boxplots for Machine Delays **117**

B Boxplots for Job Delays **123**

Overview of Generative AI Tools Used **129**
Programming . . . 129
Writing . . . 130

Acronyms **135**

List of Symbols **137**

List of Figures **139**

List of Tables **141**

List of Algorithms **143**

Bibliography **145**

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

The Scheduling Problem is an optimization problem in computer science. In general, it involves allocating resources, like machines or workers, to tasks that need to be performed. Machine Scheduling is a specific type of scheduling that involves activities taking place on a Shopfloor. The goal is to establish an order of jobs that returns an optimal value for one or more objective functions [1].

Efficient production scheduling is crucial for manufacturers in order to maintain high resource utilization, minimize delays, and keep production costs low. Induced by trends such as producing Lot-Size-1 and a shortened product-life-cycle, manufacturers are confronted with an ever-changing environment, which induces various uncertainties. Due to the dynamic environment, quick reactions to unexpected events are indispensable to ensure that resources are used as efficiently as possible [2].

The efficient usage of resources is an essential component for companies; ceteris paribus, companies with more efficient resource allocation can produce cheaper goods, leading to a competitive advantage. Production Planning helps to precisely answer: What should be produced? When and where should production take place? How much should be produced? What is the most efficient way to fulfill all of this [2]?

Smart Manufacturing and the Digitization of production systems in the context of Industry 4.0 impose various requirements on efficient scheduling algorithms. Today's scheduling algorithms need to be dynamic, fast, adaptive, and at least partially scalable to fully capture the modern dynamics and increased complexity of Smart Manufacturing. Static approaches are often insufficient for the requirements of highly dynamic Smart Manufacturing environments [3].

Multiple variants of Machine Scheduling exist: Single Machine, Parallel Machine, Flow-Shop, Job-Shop, Open-Shop. The most frequently studied variant of Machine Scheduling Problems is the JSSP. In its simplest form, there are n jobs that need to be performed on m machines, while the most common objective is to minimize the makespan (the time needed to execute all jobs) [4].

Since the Traveling-Salesman Problem (TSP) is a special case of the JSSP (the cities are the machines and the salesman is the job), and is NP-hard, the JSSP is also NP-hard [4]. There have been many ways to create approximate solutions for the JSSP (such as constraint programming, Dispatching Rules (DPR), or genetic algorithms) in the past [5]; a currently active field of research is to solve the problem with various Machine Learning (ML) approaches. Including the aforementioned dynamic real-life factors (such as machine breakdowns, delays, and new/canceled orders), one speaks of a dynamic JSSP [1].

The dynamic JSSP can be well modeled by introducing stochastic behavior into the Shopfloor environment, e.g., by simulating unexpected machine breakdowns or altering the expected job duration by an error derived from some probability distribution [6].

Another important distinction regarding JSSP is the flexibility. If jobs can be executed on more than one machine, one speaks of the flexible JSSP, which is a generalization of the JSSP. This flexibility implies an even greater search space than the classic JSSP; however, it is also more realistic for modern manufacturing environments [7].

One particularly interesting ML approach is Reinforcement Learning (RL), due to its natural ability to learn sequential decision-making under constraints based on interactions with the environment. Furthermore, no labeled data is needed (such as in supervised learning), which is scarce for the JSSP [8].

However, tabular-based RL suffers from the curse of dimensionality. With increasing complexity, the problem space becomes too large; efficient learning is hardly possible anymore. Function approximation methods replace the table with a parametrized function, which can be, for example, a Neural Network (NN). A prominent algorithm for such an approximation would be Deep Q-Networks (DQN), which improves the Q-Learning algorithm by replacing the Q-Table with a NN. For the JSSP, the type of NN used to replace the Q-Table is most often either a MLP or a CNN [9, 10].

Another improvement in RL was achieved by using multiple agents to learn an environment. For the dynamic JSSP, each machine would be an agent. Each agent separately decides which job (from the pool of available jobs) it should perform next. In the context of JSSP, the multi-agent problem is mostly modeled as cooperative and decentralized. The reward can be modeled in various ways; for example, instead of just incentivizing the makespan at the end (which implies a sparse reward), one could

additionally reward each agent with a high utilization of the machine. Each agent can learn the characteristics and behavior of their own sub-environment, leading to more generalized learning for each agent and representing a clear improvement compared to single-agent solutions, where one agent needs to learn all characteristics of the whole environment at once. While MARL introduces the problem of non-stationarity due to parallel learning, it empirically leads to faster convergence of the algorithm, yielding better results and improved scalability [11, 12].

When solving the JSSP, it is necessary to either provide the jobs as options directly or represent the relationships between the jobs and machines to the learning algorithm. Constraints such as predecessor jobs and limitations on usable machines for a job need to be represented to the learning algorithm. There are 3 options to represent the JSSP: matrix-based, statistics-based, and Graph-based. While matrix-based state representations are intuitive and easy to develop, they impose a fixed size and are not scalable, leading to a maximum number of jobs and machines on which the algorithm is trained; therefore, it is not suited for the dynamic JSSP. Statistics-based state representations can be very powerful and are scalable; however, they cannot be directly learned; they are rather handcrafted by domain experts. These limitations have led to a rising interest in Graph-based approaches, which represent the dynamic JSSP in a (often disjunctive, sometimes bipartite) Graph [12, 13].

A Graph Neural Network (GNN) often calculates so-called k-neighborhoods. Each node and each edge are represented by an n-dimensional vector. In k aggregation steps, neighborhoods are aggregated through some function for each vector, which is then stored as the new vector. Certain message-passing GNNs can be as powerful as the Weisfeiler-Lehman Graph isomorphism test (a very powerful test known to distinguish a broad class of graphs) under specific assumptions, especially if the aggregation function is injective. It is possible to further improve the expressiveness of the disjunctive Graph by using Graph Attention Networks (GATs) [14, 15].

Combining MARL with GNNs is a promising recent approach for dynamic JSSP settings. An agent whose machine is idle compares the distance between its own vector and the vector of each available node/job; the closest node based on some distance metric is chosen as the next job [13, 14].

To the best of the author's knowledge, the effect of downstream neural-network architecture in Graph-based MARL for the dynamic and flexible JSSP has not been systematically analyzed. It is not clear what size or depth the NN must have at a minimum and how this is related to the size of the JSSP instance. To the best of the author's knowledge, no work identified in the literature review systematically compares the effectiveness of MLP and CNN in this specific context. Furthermore, it is often not clearly stated how the dynamic factors of the JSSP have been instrumented. It is not clear which and how many different dynamic factors a Graph-based MARL algorithm

can learn. Both the neural network architecture and the learnable factors will be the focus of this work (see section 1.2).

1.2 Research Question and Research Objective

Solving the dynamic and flexible JSSP with MARL and GNNs is a promising approach; however, more research must be done in the evaluation of the NNs used by the agents and on the impact of different dynamic factors. This master's thesis aims to fill this gap with the following research questions:

- **RQ1**: How do the architectural hyperparameters and regularization of Graph-based MARL agents affect convergence on the flexible JSSP as the instance sizes increase?
- **RQ2**: Which of the two dynamic factors introduced can Graph-based MARL Scheduling algorithms learn?

RQ1 is split into three sub-questions:

- **RQ1.1**: How do the architectural hyperparameters and regularization of Graph-based MARL agents affect convergence when using a MLP?
- **RQ1.2**: How do the architectural hyperparameters and regularization of Graph-based MARL agents affect convergence when using a CNN?
- **RQ1.3**: How do the results differ for MLPs and CNNs?

RQ2 is split into two sub-questions:

- **RQ2.1:** What is the difference between learning machine delays and job delays?
- **RQ2.2:** What happens if both dynamic factors are learned simultaneously?

RQ2.2 can only be evaluated experimentally if both dynamic factors can be learned individually. Therefore, the combined experiment is treated as conditional: if one of the dynamic factors cannot be learned reliably in isolation, the simultaneous learning of both factors would not lead to interpretable results.

1.3 Methodology

To answer those questions, an environment that can create training data for various types of instances is necessary. A generalized base environment was created during the course 'interdisciplinary project for data science' and will be adapted for this master's thesis. The adapted version of the Shopfloor-Simulator will be able to create

training data for static, dynamic, and flexible JSSPs with different types of stochastic uncertainties. machines are modeled as finite, probabilistic, event-driven state machines [16].

For both RQ1 and RQ2, an overall architectural design similar to Park et al. [13] will be used. In this architecture mentioned, a GNN creates a job embedding. This embedding is then used to calculate similarity scores between jobs and machines with a NN. The job with the highest similarity score is then selected as an action. This type of architecture is also used in different works. Other architectures have been proposed too, but they are not the focus of this master's thesis.

RQ1 aims for an analysis of the necessary depth and hyperparameters of the NN and its correlation to convergence. In detail, for each defined JSSP instance, the network will be tested with 1 layer at the beginning, increasing to up to 5 layers and using a predefined grid of potential hyperparameters. Furthermore, a relationship between the NN architecture and the number of episodes until convergence will be analyzed.
The type of neural network differs for RQ1.1 and RQ1.2. After testing both MLPs and CNNs, the results will be compared. This comparison aims to detect differences in convergence, suitability for different types of instances, and limitations of the given NN.

The agents are deployed in a cooperative multi-agent setting, where each machine is represented by one agent instance. The implementation follows a centralized shared-policy MARL design: a central coordinator maintains the global Shopfloor state, determines the currently executable jobs, manages the replay buffer, and coordinates the training process. The agent instances share a common neural network model and therefore apply a shared policy conditioned on the current machine state and the candidate job embedding.
As a reward function, the objective function suggested by Park et al. [13], leveraging a normalized reward as an intermediate reward, will be tested.
A set of predefined features for jobs and machines will be used to create the representations of jobs/machines. RQ1 will be tested on flexible JSSPs; no dynamic factors will be used.

RQ2 analyzes the convergence of different dynamic factors. Each dynamic factor will be tested individually (RQ2.1). Afterwards, the original intention was to combine both dynamic factors and analyze their joint effect (RQ2.2). However, this combined experiment is only meaningful if both dynamic factors can be learned individually. If this prerequisite is not fulfilled, RQ2.2 is answered by explaining why the simultaneous learning of both factors cannot be interpreted in a scientifically meaningful way.
The results from RQ1 will serve as guidance for the architectural design of the NNs. Furthermore, only a subset of objective functions will be used for RQ2; the exact subset depends on the results from RQ1.
It is possible that the proposed architectures resulting from RQ1 may need to be refined in terms of size and/or the hyperparameters used. If an adaptation is necessary, an analysis regarding the increased complexity induced by the given dynamic factor will be

performed. RQ2 will be tested on the dynamic and flexible JSSP.

The 2 dynamic factors to be analyzed are:

- job delays caused by machines
- job delays caused by jobs

Learning is evaluated by analyzing makespan and utilization. If the makespan decreases and stabilizes, this is interpreted as evidence that the algorithm has adapted to the problem instance. If the makespan is lower than simple dispatching rules such as First-In-First-Out (FIFO) and Shortest-Processing-Time (SPT), the algorithm performs better than these basic rule-based baselines in the tested setup.

1.4 Scientific Relevance

1.4.1 Scientific Relevance

The JSSP is not only applicable to Smart Manufacturing; the essence of this NP-hard problem has applications in various other domains. Instance types can also be revolutionized in other domains. The results may provide initial indications for architecture selection in related graph-based MARL scheduling settings, but their transferability to other JSSP variants requires further validation. Furthermore, by differentiating between the machines' perspective and the jobs' perspective, other features and attributes related to the core problem are revealed.

1.4.2 Practical Relevance

Solving the dynamic JSSP in a practical manner is an essential challenge in Smart Manufacturing. Developing practical solutions can imply reduced downtime, support lean manufacturing, strengthen resilience, and help make real-time adaptations to machine allocations. All these implications would lead to increased revenue for companies applying such techniques.

CHAPTER 2

Fundamentals

This section covers all foundational topics needed for this thesis. At the beginning, each of the five topics will be examined individually. While some topics partially build on each other or can be viewed as subsets of other topics, cross-references are kept to a minimum. The five foundational topics are:

- Smart Manufacturing And Industry 4.0 (section 2.1)
- Scheduling Problems (section 2.2)
- The Job-Shop Scheduling Problem (section 2.3)
- Reinforcement Learning (section 2.4)
- Graphs (section 2.5)

Afterwards, those topics will be concatenated into one framework in section 2.6.

2.1 Smart Manufacturing and Industry 4.0

In this section, the focus is on 'Smart Manufacturing' and its relation to the term 'Industry 4.0'. After defining the necessary terminology and covering the historical background, the core technologies enabling the transition towards Industry 4.0 are identified. The well-known Automation Pyramid is explained in a separate section together with its limitations in Smart Manufacturing. To prepare for the content of the next section, an overview of how Production Planning and Control (PPC) works and how it is often implemented is given at the end of this section.

2.1.1 Smart Manufacturing

Industry 4.0 is a new paradigm transforming the industry by introducing new technologies in the production process. A non-exhaustive list of examples of such technologies includes the Internet-of-Things, Industrial Cyber-Physical Systems, Artificial Intelligence (AI), and Cloud Computing [17].

The term 'Industry 4.0' was first introduced within a strategic program of the German government in 2009. In that context, Smart Manufacturing is often interpreted as the actual implementation of this strategy. The initial focus of Smart Manufacturing was mostly on process automation and supervision. However, over time, more and more technologies outside the area of pure automation started to play significant roles. A non-exhaustive list of such technologies would include Cyber-Security, Remote Monitoring, and Digital Twins [17].

Figure 2.1 shows the industrial revolutions:

- The **First Industrial Revolution** was introduced by the invention of the steam engine in the late 18th century
- The **Second Industrial Revolution** started with the beginning of the 20th century and was mainly built on the utilization of electricity as an enabler for mass production and assembly lines.
- The **Third Industrial Revolution**, sometimes also referred to as the Digital Revolution, started in the 1970s and was characterized by automation techniques through information technology.
- The **Fourth Industrial Revolution** and the **Fifth Industrial Revolution** may not be viewed as distinct Revolutions; rather, Industry 5.0 is driven by the limits of natural resources on planet earth, strengthening people, sustainability, and resilience within the strategic framework of Industry 4.0.

The necessity for this digital transformation is also often argued through consumer trends. While the trend in the 20th century shifted towards mass production and standardization of products, the focus has shifted back towards individualized products in the 21st century. This trend towards a Lot-Size of 1 is not only a challenge for manufacturing companies but is also viewed as an opportunity, as margins in personalized products tend to be much higher. This individualization necessitates higher flexibility at all levels of the production cycle compared to legacy systems. Additionally, Smart Manufacturing provides an opportunity for more cost-efficient production. A higher level of automation and control over the production cycle enables optimized resource utilization, reduced downtime through predictive maintenance, faster response to market changes, and consistent product quality with minimal waste [17].

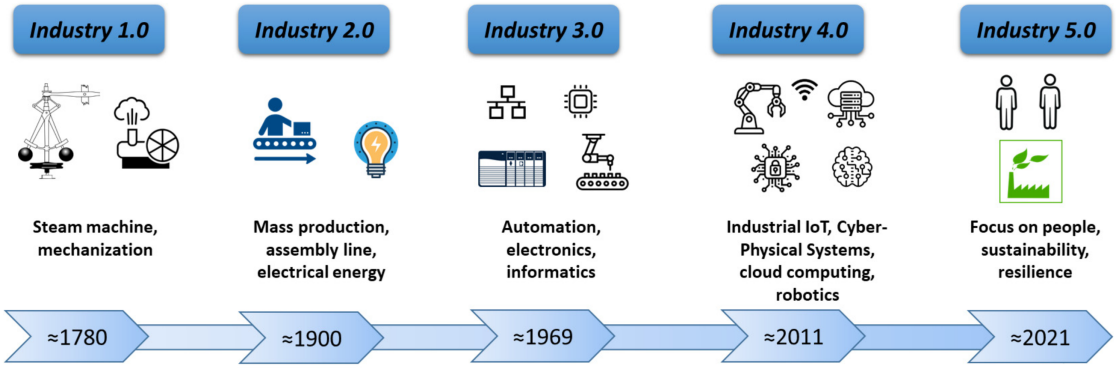


Figure 2.1: Timeline of industrial revolutions from Industry 1.0 to Industry 5.0 according to [17]

2.1.2 Enabling Technologies

Most often, nine technologies (listed below) are viewed as enablers for Smart Manufacturing:

- Industrial Internet of Things
- Cyber-Security
- Vertical and horizontal System Integration
- Collaborative Robots
- Big Data and Artificial Intelligence
- Virtual/Augmented Reality
- Additive Manufacturing
- Simulation
- Cloud Computing

An interesting perspective is that Industry 4.0 has emerged as a response to these new technologies and their expected effects when integrating them into Automation Systems.

2.1.3 Automation Pyramid

An architecture often referred to is the Automation Pyramid shown in figure 2.2: *Level 1* captures process data acquired through various types of sensors within the process. *Level 2* includes logical control systems or devices executing control algorithms based on the information provided from Level 1.

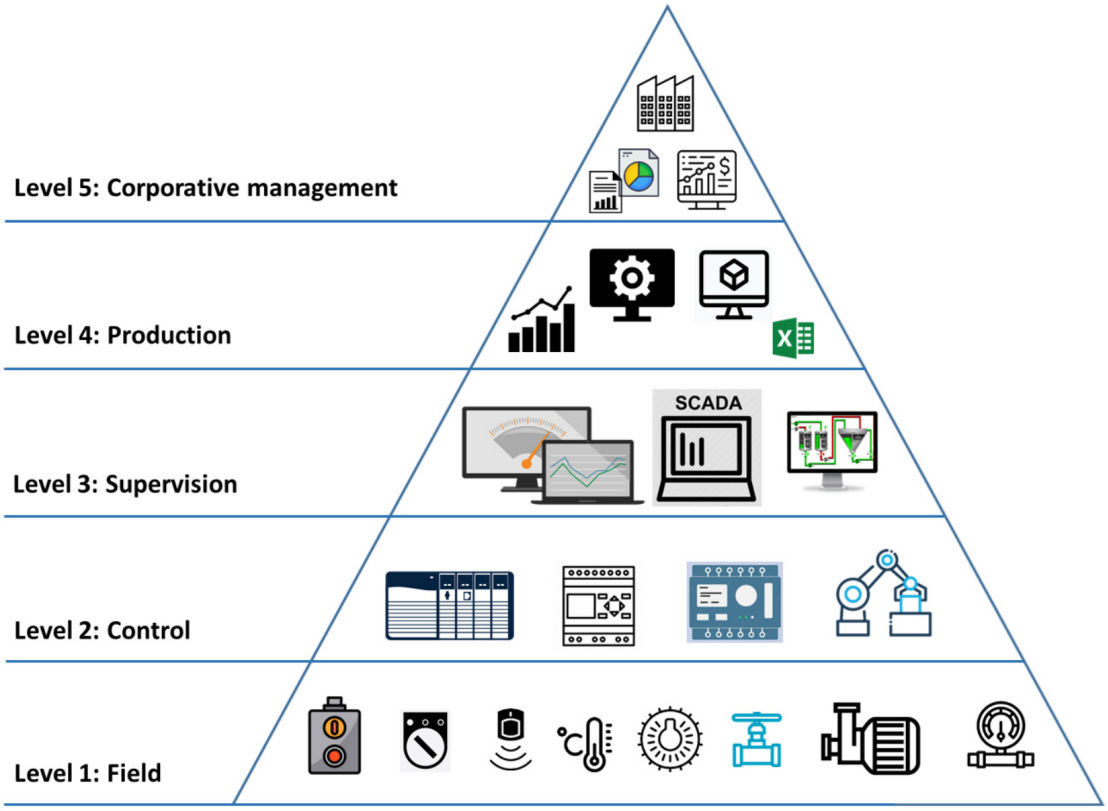


Figure 2.2: Automation Pyramid

Level 3 describes the supervision of all processes performed in the two lower levels.
Level 4 incorporates the monitoring of all processes managed, including transportation, quality control, and maintenance.
Level 5 is the Corporate Management planning at a strategic level.

The Automation Pyramid is commonly referred to in the literature and is well established. However, this model has been challenged in the last few years, with criticisms of its hierarchical structure, as this implies that not all layers are connected. Critiques of the Automation Pyramid argue that the lack of connectedness between multiple layers leads to a lack of information, efficiency, and potentially poor decision-making. Different architectures incorporating the current trends and requirements of Smart Manufacturing have been proposed. Regardless, no other architecture is as established as the Automation Pyramid yet [17].

2.1.4 Production Planning and Control

PPC includes the activities of loading, scheduling, sequencing, monitoring, and controlling the use of resources and materials during production. *Loading* defines 'how much to do',

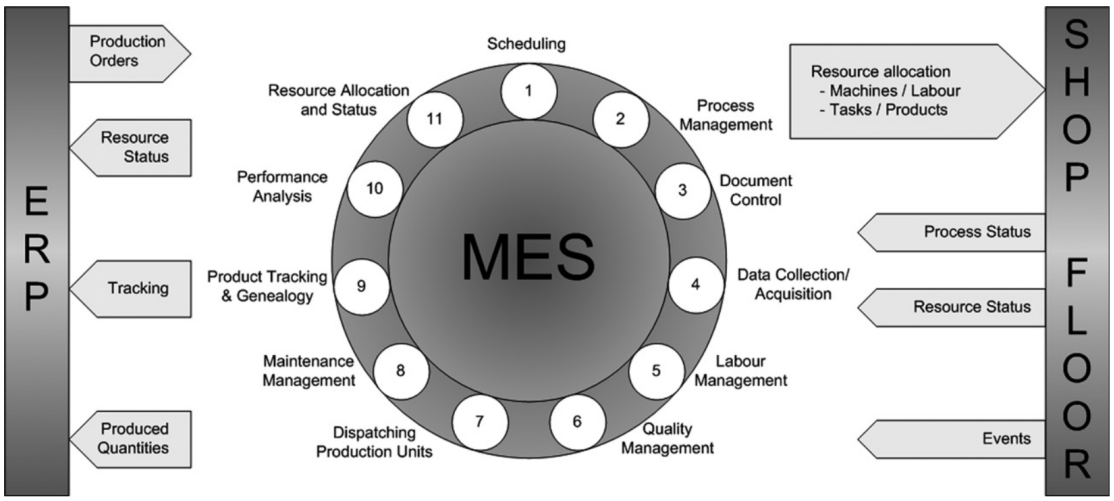


Figure 2.3: MES functionalities according to [19]

while *scheduling* defines 'when to do it'. For example, scheduling defines start times and end times for jobs to be executed. In contrast, *sequencing* determines 'in what order', i.e., the order of the jobs to be done. *Monitoring* and *control* refer to measuring the activities, evaluating whether those activities are performed as expected, and, if necessary, setting corrective actions [18].

In essence, PPCs manage uncertainty in production systems. This can be done through stabilization techniques, prediction, or efficient and effective reactions to real-time events. Various smart technologies already mentioned in the last section played a big role in the improvement of PPC systems over the last few years [18].

Historically, Enterprise Resource Planning (ERP) systems have been coordinating PPC activities. Due to their lack of real-time decision-making, Manufacturing Execution Systems (MES) and Advanced Planning and Scheduling (APS) systems have been developed since the early 21st century. Whereas MES monitors the Shopfloor in real-time between planning and actual execution, APS systems create the planning for the Shopfloor. In this context, the Shopfloor refers to the production area within a manufacturing facility where machines, operators, and workstations are located, and where the actual transformation of raw materials into finished products takes place [18].

The functionalities of the MES are shown in figure 2.3. Driven by the demands (from the industry and the authorities) for reactivity, quality, respect for standards, reduction in costs, and deadlines, it serves as an interface between the ERP and the Shopfloor [19].

In contrast to the MES, APS systems, as shown in figure 2.4, emphasize the planning and scheduling tasks of the Shopfloor in different time horizons. Whereas APS systems create plans and schedules, the MES ensures that those plans are executed as closely as

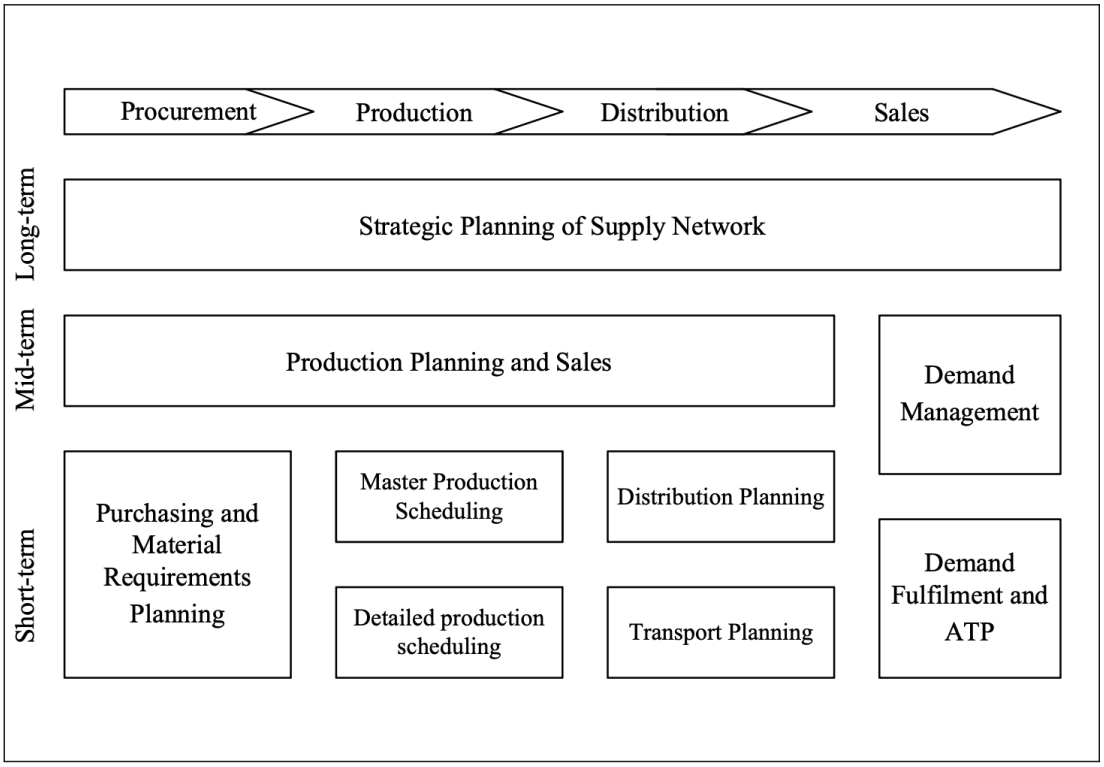


Figure 2.4: APS functionalities according to [20]

possible according to the planning by the APS system. In essence, the APS system sets the target, while the MES aims to support the fulfillment of the target through tracking and adjusting events in real time [20].

Nowadays, production systems generate an increasingly large volume of data, enhancing the potential to use this data for improving performance and shortening production cycles. The applications for using that data in PPC can be categorized into the following three levels:

- *Strategic*: long-term, such as business development
- *Tactical*: medium-term, such as demand-forecasts
- *Operational*: short-term, such as predicting maintenance for a specific machine

PPC operates on all of those levels. Especially at the tactical and operational levels, the use of big data and ML has developed rapidly over the last few years. While ML techniques can also be used for strategic planning, e.g., forecasting, the other two levels of planning have seen a greater increase in applications due to their higher level of automation. At the tactical and operational levels, the environment in which ML

techniques are to be applied can be (numerically) described more precisely compared to more abstract concepts like customer satisfaction. Choosing the appropriate ML technique is highly case-specific and is often not generalizable, as it also depends on the level of digitization [18].

2.2 Scheduling Problems

Scheduling, as one element of PPC, is one of the areas that saw a huge increase in ML-applications over the last decade. Scheduling in general involves the allocation of resources (such as machines, workers, or transportation assets) to tasks over a given time period.

2.2.1 Historical Context

The first application for Scheduling was the GANTT-Chart introduced in the early 20th century. Since the 1950s, when the first literature regarding the Two-Machine flow shop was published, Scheduling theory has emerged as an independent field of study within operations research [21].

From the 1950s to the 1970s, Scheduling was modeled as a mathematical optimization problem, which was later classified as NP-hard due to its combinatorial nature. An NP-hard problem is a computational problem for which no known polynomial-time algorithm exists. The focus on solving the scheduling problem then shifted from exact solutions to approximate solutions, which can be solved in a reasonable time and, depending on the approximation, guarantees a good (but not optimal) schedule.

With the increasing utilization of personal computers in the 1980s, Scheduling has developed even more rapidly through increased computing resources. Different variations of the Scheduling Problem also arose during the change of the manufacturing paradigm in recent years, such as flexible or dynamic scheduling, or distributed multi-factory scheduling [21].

2.2.2 Classification

The number of different Scheduling solutions is in the hundreds; a broad categorization of those solutions is given in figure 2.5:

- *Mathematical Programming Methods* cover methods that solve the Scheduling Problem in an exact fashion. Whether exact methods can be used at all depends on the problem size and sufficient modeling of the environment.
- *Heuristic Algorithms* cover problem-specific rule-based algorithms that are often based on the experience of the specific problem. They do not solve the Scheduling Problem exactly; they trade off solution optimality for computational efficiency.

Heuristic algorithms can (in most cases) also solve larger problem sizes efficiently. In some cases, they are also able to overcome insufficient modeling of the environment, as its behavior is often determined by domain experts, which closes the knowledge gap.

- *Meta-heuristic Algorithms* can be viewed as generalized heuristic frameworks specialized in escaping local optima. Often inspired by nature (see genetic algorithms), they escape local optima through stochastic behavior in a reasonable time while avoiding expensive computations. Meta-heuristic algorithms are commonly used in scheduling tasks because of these features. However, they only provide an approximate solution and often do not have the approximation guarantees needed to evaluate how good the scheduling actually is.

- *Simulation methods* model and simulate production environments (or Shopfloors) under certain conditions and derive solutions in complex and often stochastic scenarios.

- *AI Methods* use ML techniques to learn Scheduling policies. AI methods can, at least theoretically, solve all issues mentioned for the other solution categories. However, they also require a lot of good quality data, often requiring resource-intensive training, and come with a lack of interpretability (black box). The first drawback can be mitigated by a certain level of Digitization of the Shopfloor. The lack of interpretability remains an open research area in the domain of AI.

The author of this master's thesis argues that the distinction between simulation methods and AI methods is blurred and not clearly separated, as AI methods may also learn from simulations.

2.2.3 Machine Scheduling: Different Variations and How to Solve Them

One type of Scheduling is machine Scheduling, where the resources to be allocated are machines or machine-like resources, such as workstations. Most often, the resources are utilized by jobs, which can be, for example, one process step. machine Scheduling Problems are also solved through an objective function under certain constraints. It can be separated into the following categories [1]:

- **Single Machine**: more than 1 job needs to be executed on 1 machine. This simple problem may also arise naturally in Shopfloors if one machine is considered a bottleneck.

- **Parallel Machine**: more than 2 jobs need to be executed on exactly one of two machines. The time needed for job execution may be identically (or uniformly) distributed, or different.

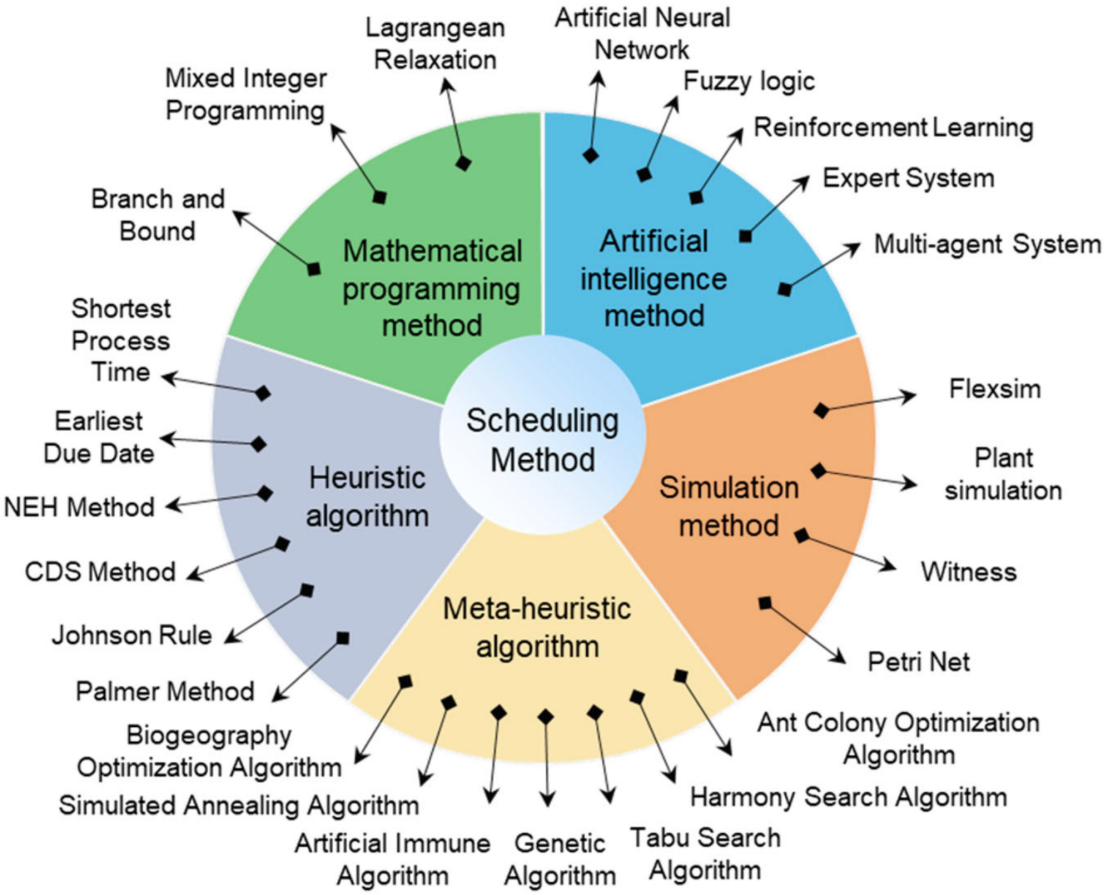


Figure 2.5: Scheduling approaches according to [21]

- **Flow-Shop**: A set of jobs must iterate through the same set of machines in a given sequence.
- **Job-Shop**: Like the flow-shop; however, each job may have its own defined route through the Shopfloor.
- **Open-Shop**: Like the job-Shop, but without a constraint on the processing order of each job.

2.2.4 Current State in the Industry

Most often, the actual machine Scheduling algorithms in the industry are simpler than the advanced and sophisticated solutions proposed by recent literature. Simple algorithms such as FIFO, together with a human feedback loop and potential intervention, are the most common approach. While there are regional differences in the level of advancement of Scheduling algorithms and in the digitization of the factories, the reasons for the

absence of sophisticated machine Scheduling algorithms can be roughly split into the following three arguments [22]:

- **Legacy Systems & Limited Progress in Digitization**: The Shopfloor cannot provide the necessary information in real-time. Especially in factories built before Industry 4.0 was invented, the costs of renewing the factories to the provided level of digitization if often not economically justifiable [22].
- **Real-World Complexity**: The assumptions made in academia are often too oversimplified. Moreover, many studies focus on theoretical problems instead of solving actual Shopfloor issues [22, 23].
- **Simplicity**: Simple algorithms such as FIFO can be well understood by humans, whereas complex algorithms, especially when combined with NNs, are mostly viewed as a black box [23].

Bridging this gap between industry and academia remains an open challenge that will need to be tackled during the transition towards Industry 4.0.

2.3 The Job-Shop Scheduling Problem

The JSSP involves the assignment of jobs to machines under the constraint that one machine can only process one job at a time. The most common objective used for the JSSP is the makespan; arguments for this are provided throughout this section. The JSSP is not only NP-hard, but it is one of the hardest combinatorial problems in general. As an example: The well-studied TSP is also NP-hard, but it is a simple special case of the JSSP (the salesman would be the only machine selecting its jobs/cities and finishing as fast as possible) [4].

2.3.1 Problem Statement

Let one JSSP instance be defined as follows:

- Let $\mathcal{O}$ be the set of **orders**.
- Let $\mathcal{M}$ be the set of **machines**.
- Let each **order** $o \in \mathcal{O}$ **consist of** a sequence of n **jobs** $J_{o1}, J_{o2}, \ldots, J_{on}$.
- Let each **job** J_{oi} have a **processing time** $p_{oi} \in \mathbb{R}^+$.
- Let each **job** J_{oi} be processed on **one specified machine** $m_{oi} \in \mathcal{M}$
- Let the **start time** of job J_{oi} be denoted as $s_{oi} \in \mathbb{R}_{\geq 0}$.
- Let the **completion time** of job J_{oi} be denoted as $c_{oi} = s_{oi} + p_{oi}$.

- **Order Precedence**: jobs of the same order must follow their given sequence: $s_{o,i+1} \geq c_{oi} \quad \forall\, o \in \mathcal{O},\ i < n_o$
- **machine capacity**: Each machine processes at most one job at a time: if $m_{oi} = m_{o'i'}$ for two distinct jobs, then $s_{oi} \geq c_{o'i'} \quad \text{or} \quad s_{o'i'} \geq c_{oi}$.
- The **makespan** is defined as $c_{\max} = (\max_{o\in\mathcal{O}}\ \max_{k\in J_o} c_{oi}) - (\min_{o\in\mathcal{O}}\ \min_{k\in J_o} s_{oi})$
- A **Schedule** σ is a mapping $\sigma : (o, i) \mapsto s_{oi} \in \mathbb{R}_{\geq 0}$ that satisfies all precedence and machine capacity constraints.
- An **optimal Schedule** σ^* is one that minimizes the makespan: $\sigma^* \in \arg\min_{\sigma\in\mathcal{F}} c_{\max}(\sigma)$, where $\mathcal{F}$ is the set of all feasible Schedules.

Without considering any further constraints (for example, energy efficiency), every optimal solution must minimize the makespan [4]. When talking about one specific instance of the JSSP, the problem size is defined as $m \times n$, where m denotes the number of machines and n denotes the total number of jobs.

2.3.2 Dynamic JSSP

The definition of the JSSP according to section 2.3.1 suits a static JSSP, that is, a non-stochastic environment. Dynamic JSSPs introduce various types of uncertainties; a non-exhaustive list of uncertainties is as follows:

- Uncertainty in processing time p_{oi}
- New job arrivals
- Machine Breakdowns
- Due date uncertainty (in a realistic setting, orders have due dates, which may change due to external factors)

A deviating processing time and machine breakdowns are the two most commonly studied uncertainties in the literature. However, the definition of the dynamic JSSP is not consistent in that regard and varies across different literature. Especially with increased computational resources and a trend towards Smart Manufacturing, dynamic JSSPs have been studied more frequently in the last decade. This can also be argued with a smaller lot size in many production environments, which induces dynamic JSSPs to be more practically relevant than in the 20th century [6].

While the dynamic JSSP is a natural extension leading to a more realistic setup through the introduction of uncertainty, it presents a different and even more complex problem statement. The search space of this problem statement can be viewed as a decision tree, where each dynamic event branches the search space into new scenarios. The static JSSP is exponential with respect to the problem size. The dynamic JSSP is exponential with respect to the problem size, multiplied by the number of dynamic events [24].

2.3.3 Flexible JSSP

The flexible JSSP relaxes the constraint that each job can only be processed on one machine. I.e., m_{oi} becomes a subset of $\mathcal{M}$: $m_{oi} \subseteq \mathcal{M}$. This trend towards increased flexibility in machine assignment is also induced by current trends in Smart Manufacturing: during mass production in the third industrial revolution, machines were often specialized for processing one type of jobs (or at least a very narrow subset). Nowadays, machines can often process different types of jobs. Workstations, which also fall under the definition of a machine in that context according to section 2.2.3, are most often also able to process different types of jobs [1].

Similar to the dynamic JSSP, making the JSSP flexible yields a problem definition that is more related to the requirements for Smart Manufacturing, but it further increases the complexity of the search space. For a JSSP instance where every job can be executed on every machine, the search space grows exponentially with respect to the number of jobs.

2.3.4 Solution Types for the JSSP

There are two different solution types for the JSSP: partial-based solution algorithms and whole-based solution algorithms.
Whole-based algorithms provide a solution for the complete Schedule at once. This means that such algorithms return, at time step t, a given Schedule for all available jobs at once. An example of such an algorithm would be a constraint solver [12].
In contrast, *partial-based algorithms* only yield parts of the Schedule at time step t. Instead of Scheduling all jobs at once, only a subset of the jobs is assigned a machine at time step t. The magnitude of this part for each time step is dependent on the given algorithm. An example of such a type would be genetic algorithms [12].

2.3.5 Other Extensions of the JSSP

Making the JSSP dynamic and/or flexible is not the only studied extension of the problem statement. For example, the distributed JSSP incorporates that jobs may be executed at different locations. Focusing on one Shopfloor, one may also include transition times between machines, workers who need to preprocess and/or postprocess jobs on the machine, or machines that can process multiple jobs at a time. As those extensions are not the focus of this thesis, they are not examined further.

2.3.6 Objective Functions

Makespan is the most common type of objective function used for the JSSP. However, other optimality criteria may also be defined additionally [1]:

- **Lateness**: $L = \sum_{o \in \mathcal{O}}(c_o - d_o)$, where o denotes an order, d_o denotes the due date of the order, and c_o denotes the completion time of the order.

- **Tardiness**: $T = \sum_{o \in \mathcal{O}} max(0, L_o)$. In this case, early completions of the order count as zero. One may also define Tardiness as a maximum delay: $\max_o T_o$
- **Earliness**: $E = \sum_{o \in \mathcal{O}} max(0, -L_o)$. In this case, the number of completions before their given due dates is emphasized.
- **Flow Time**: $F = \sum_{o \in \mathcal{O}} (c_o - r_o)$, where r_o denotes the release time of an order (the earliest time its first job can start)
- **Waiting Time**: $W = \sum_{o \in \mathcal{O}} (F_o - \sum_{k \in J_o} p_{ok})$, where the goal is to increase resource utilization by decreasing the waiting time.

Objective functions different from makespan can serve two purposes:

- If the solution type according to section 2.3.4 is partial and the main goal is to optimize makespan, it is sometimes necessary to evaluate a schedule at time t, where $t \neq T$ (T equals the final timestep of creating the schedule). At this timestep, the final makespan is not known yet, and the current makespan may be deemed too uninformative. Other evaluation criteria, such as waiting time, often have an (inverse) relationship with the makespan and can be used as an approximation.
- On the other hand, if the objective of the JSSP is not solely focused on makespan but also includes, e.g., penalties for late orders or energy consumption, the objective function often consists of a composite objective function, mixing e.g., makespan, lateness, and energy consumption into one objective function.

2.4 Reinforcement Learning

This section covers the centerpiece of this thesis. General explanations are provided at the beginning, starting with simple, tabular-based methods.
After enlightening the core principles and concepts developed in the 20th century, the focus shifts towards modern approaches that often involve some sort of function approximation to partially solve the curse of dimensionality and enable generalization (i.e., Deep Reinforcement Learning (DRL)).
With a few selected milestones, the goal is to understand the most influential factors that have led to architectures that surpass the performance of human experts in some domains.
Afterward, the focus shifts to multi-agent Reinforcement Learning, an extension in which multiple agents are deployed.
Theoretical validations are provided if possible; emphasis is placed on differentiating between models that are proven to converge to the optimum under certain conditions and models that yield good results based on empirical validation.

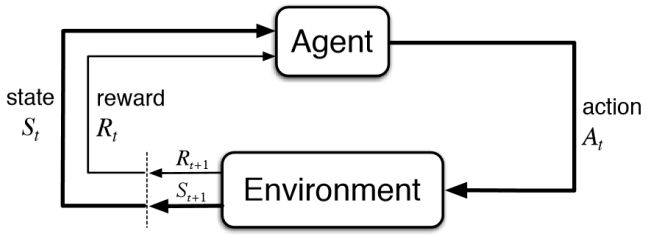


Figure 2.6: Interaction between agent and environment according to [25]

2.4.1 Introduction

Reinforcement Learning (RL) is based on the idea that human learning is induced by interactions with the environment. It maps situations to actions in order to maximize a long-term reward. One must discover which actions yield the most reward in the long term [25].

The two most important features that distinguish RL are a trial-and-error search and a delayed reward. This is in contrast to unsupervised learning, where the identification of clusters is the focus, and to supervised learning, which learns from patterns in the data [25].

RL is often formalized as the optimal control of an incompletely known Markov Decision Process (MDP):

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R)$$

where:

- $\mathcal{S}$ is the set of states,
- $\mathcal{A}$ is the set of actions,
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function, where $P(s' \mid s, a)$ gives the probability of transitioning to state $s^{'}$ when taking action a in state s,
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, where $R(s, a)$ is the expected immediate reward received after taking action a in state s

If the next state depends only on its current state and action, it is said to have the Markov Property.

A (learning) agent interacts with an environment: at time step t, the agent transmits an action a_t in a given state s_t to the environment, which responds with a new state s_{t+1} and a reward signal r_t; see figure 2.6.

The learner wants to maximize the expected return (= the sum of the rewards). The reward must signal to the agent what needs to be achieved (not how) [25].

The interaction is happening as a sequence of discrete time steps[1]:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, ... \tag{2.1}$$

2.4.2 Episodic and Continuous Tasks

Certain types of tasks have a defined terminal state and are therefore finite, such as the board game chess. Each game of chess may be viewed as one episode. Such episodic tasks have a set of non-terminal states, with one or more paths towards a terminal state. The set of all states, including the terminal state, is defined as S^+. The return may be formulated as in formula 2.2, where G_t, the expected return at time step t, is the sum of the expected rewards of the future (discrete) time steps until the final time step T [25].

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{2.2}$$

Other environments do not naturally break down into episodes but are continuous, such as a robot moving its arm. Such continuing tasks often use a discounted expected reward as shown in formula 2.3. Discounting must be used for continuous tasks because, otherwise, the expected reward could easily be infinite [25].

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.3}$$

2.4.3 Value Functions and Policies

Almost all RL algorithms involve estimating value functions, which map states (or state-action pairs) to the expected return. A policy is defined as a mapping from states to the probabilities of selecting each possible action. The value function of a state s under a policy π is denoted as $v_\pi(s)$. It is the expected return when starting in s and following π thereafter [25].
For MDP, v_π can be defined as:

$$v_\pi(s) = \mathbb{E}[G_t | s_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s], \quad \forall\, s \in S \tag{2.4}$$

Similarly, the value of taking action a in state s under a policy π (= the expected return starting from s, taking the action a, and thereafter following policy π), is defined as the action-value function[2] $q_\pi(s, a)$:

$$q_\pi(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a], \quad \forall\, s \in S \tag{2.5}$$

[1] Although approaches for continuous time cases exist, this is not the focus of this thesis
[2] The action-value function is often referred to as Q-Function or quality-function

If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $a_t = a$ if $s_t = s$.

Solving a RL task optimally is therefore finding an optimal policy. For finite tasks, a policy is optimal if $v_{\pi^*}(s) \geq v_{\pi'}(s) \quad \forall\, s \in S$, where π^* denotes the optimal policy and π' denotes any other policy. There may be more than one optimal policy; the set of all optimal policies is defined as:

$$v_*(s) = \max_{\pi} v_\pi(s) \tag{2.6}$$

An optimal policy must equal the expected return for the best action from that state, which is formalized in the Bellman optimality equation:

$$v_*(s) = \max_{a} \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{2.7}$$

$$q_*(s, a) = \max_{a} \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} v_*(s', a')] \tag{2.8}$$

This is valid for both episodic and continuing tasks. In the case of episodic tasks, the reward is always zero after the terminal state is reached.

2.4.4 Exploration vs. Exploitation

When designing RL algorithms, the trade-off between exploration and exploitation is a key decision. If an algorithm decides on un(der)explored actions, the agent may find a new valuable state (or action in a given state). On the other hand, exploiting the currently best-known actions leads to a known high reward. The exploration-exploitation dilemma is still unresolved for a lot of problem statements; designing a reasonable trade-off is a key aspect in designing RL algorithms [25].

2.4.5 Generalized Policy Iteration

The Generalized Policy Iteration (GPI) describes two simultaneous and interacting processes, namely policy evaluation (estimating how good a policy is) and policy improvement (making the policy better based on the value estimates). The key idea is that iteratively improving a policy using value estimates will eventually converge to an optimal policy [25].

2.4.6 Monte Carlo Methods

Assuming incomplete knowledge or the inability to accurately model the environment (which is often the case in real-world applications), Monte Carlo Algorithm (MC) methods are a popular way to estimate value functions. They don't require exact models of the environment; rather, they learn the probability distributions from experience through repeated sampling of states, actions, and rewards from actual or simulated experience [25].

In RL, MC methods refer to methods based on averaging complete returns. The concept of GPI is used in MC methods by alternating between evaluating the policy using averaged returns from sampled episodes and improving the policy greedily with respect to the estimated values. The empirical average is able to shift the policy towards optimality [25].

MC methods further have the advantage that regions of interest can be sampled more often. The fourth advantage[3] is that they may be less harmed by violations of the Markov Property because they do not update their value estimates based on the value estimates of the successor state. MC methods in RL are therefore not biased [25].

2.4.7 Temporal Difference Methods

Temporal Difference Algorithm (TD) learning is similar to MC learning, as it also learns directly from raw experience. However, it introduces a constant step size parameter, formalized in:

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(S_{t+1}) - V(t)] \tag{2.9}$$

The value of state s_t is updated by the scaled difference between the originally expected value and the actual return. The scaling factor α is often called the learning rate.

By design, MC methods must finish the episode to update their value functions:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{2.10}$$

In contrast, TD methods can update their parameters during the episode because their value updates are partially based on existing estimates (i.e., they bootstrap):

$$v_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi S_{t+1}|s_t = s] \tag{2.11}$$

TD methods can therefore also be applied straightforwardly to continuous tasks. The number of (future) time steps used from the experience can be parameterized in TD methods. This parameter may be called λ.

The rationale for TD learning is that adjustments to the updated state should be based on current knowledge. As a real-life example: You are driving a car that you believe will take 30 min. After a while, you realize you have lost some time because of traffic. It makes sense to partially update your new estimation of the remaining duration based on your previous belief [25].

On-/Off-Policy Learning and Its Relation to Online/Offline Learning

In on-policy methods, the agent learns the value of the policy it is currently following. The policy to be evaluated/improved is the same policy taking actions in the environment. on-policy algorithms tend to be more stable and are well suited for learning while

[3] together with the benefits that they learn directly from the environment, can be used with simulations, and may sample regions of interest

interacting with the environment in real time, due to their simultaneous acting and learning. Methods that learn and update their parameters during interaction with the environment are referred to as online learning algorithms [25].

In contrast, off-policy methods use one policy for taking actions while evaluating/improving a different policy. This separation leads to greater control of exploration and exploitation[4]. Furthermore, as the policy to be evaluated/improved is different from the one taking actions, one can naturally train off-policy algorithms based on past experience. Methods which learn and update their parameters from past experience are referred to as offline learning algorithms [25].

A popular on-policy algorithm would be the State-Action-Reward-State-Action Algorithm (SARSA):

Algorithm 2.1: SARSA (on-policy TD control) for estimating $Q \approx q_*$ according to [25]

Data: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $q(terminal, \cdot) = 0$;
foreach *episode* **do**
    Initialize S;
    Choose a from s using policy derived from Q (e.g., ε-greedy);
    repeat
        Take action a, observe r, s';
        Choose a' from s' using policy derived from Q (e.g., ε-greedy);
        $q(s, a) \leftarrow q(s, a) + \alpha \left[r + \gamma q(s', a') - q(s, a) \right]$;
        $s \leftarrow s'$;
        $a \leftarrow a'$;
    until s *is terminal*;
end

For off-policy, a popular algorithm is Q-learning:

The difference in the algorithms 2.1 and 2.2 is how they update their value estimates; while SARSA learns the action-value function for the policy it executes (including its ϵ-greedy behavior), Q-learning, on the other hand, learns the optimal action-value function (independent of the current policy being followed). Both algorithms proved to be great assets in their given contexts.

[4]The policy used for decisions can be, for example, ϵ-greedy (taking random actions in a given fraction of decisions to be made) with $\epsilon \in [0, 1]$

Algorithm 2.2: Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$ according to [25]

Data: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $q(terminal, \cdot) = 0$;

foreach *episode* **do**

  Initialize s;

  repeat

    Choose a from s using policy derived from Q (e.g., ε-greedy);

    Take action a, observe r, s';

    $q(s, a) \leftarrow q(s, a) + \alpha \left[r + \gamma \max_a q(s', a) - q(s, a) \right]$;

    $s \leftarrow s'$;

  until S *is terminal*;

end

replay buffer

A replay buffer is essential for off-policy algorithms. The experience collected through past interactions with the environment is stored as tuples of the form $(s_t, a_t, r_{t+1}, S_{t+1}, d)$. One tuple is often referred to as a sample. d denotes a boolean flag indicating whether the terminal state was reached. After regular intervals (e.g., after an episode), the value function of the policy is updated based on samples from the replay buffer. When solving the JSSP with MARL, S is the machine State of one machine, and A is the selected job.

The characteristics (along with their benefits) of a replay buffer are [26]:

- **Update from random samples**: A certain batch size is sampled randomly, leading to an uncorrelated batch for training (improving stability).
- **Experience reuse**: Each interaction is stored and can be replayed multiple times, improving sample efficiency compared to online learning.
- **Learning from past mistakes and successes**: Important events can be revisited more often, leading to a higher chance of convergence in sparse reward or high-variance environments.

Different challenges (or design decisions) arise when working with replay buffers, such as buffer size and sampling techniques.

2.4.8 Deep Reinforcement Learning

Introduction

Until now, RL approaches explained in this section have been referred to as tabular-based methods. Such methods map each possible state-action pair to an estimated value. In practice, the state space is often combinatorial and enormous; problems for tabular-based methods arise regarding the memory needed to store such tables, as well as the data and time required to fill them. Function approximation methods use a parameterized function instead of a table to estimate value functions or policies, enabling the applicability of RL in settings with much larger state spaces. The output of the function approximation serves as input instead of the table. Since the latest AI-Boom, NNs are often used as function approximators. When using a NN as an approximator, such methods are referred to as DRL. Despite decades of research, several aspects of why NNs work so well are not yet fully understood. Examples of such aspects include the optimization dynamics and why attention scales so effectively. [27, 28].

Using DRL enables scalability and generalization. This recent approach further showcases improved results with sparse reward structures and can lead to an end-to-end training pipeline. However, partially induced by the fact that NNs are not yet fully understood, they lack the theoretical guarantees introduced earlier in this section and may not converge. Furthermore, they are sensitive to hyperparameter tuning and lead to a black box in terms of decision-making. To understand the decisions made, one needs to analyze the weights and activations inside the NN; such an analysis is currently limited and remains an open research area [28].

The Atari-Project

A major breakthrough in DRL was the DeepMind Atari-Project, which was able to learn various Atari games through high-dimensional raw pixels by using a CNN as an approximator. As this approach used a variant of the Q-Function explained in section 2.4.3, it was called Deep Q-learning (DQN). Using a replay buffer to break sample correlation and a target network for improved stability can be viewed as the two innovations of this project [27].

AlphaZero

In contrast to the off-policy approach in the paragraph above, Google DeepMind released an on-policy approach with AlphaZero, entirely based on self-play, 4 years later. Instead of approximating the Q-Function directly, the NN used in AlphaZero estimates the policy and the value separately. Instead of the ϵ-greedy approach in DQNs, AlphaZero guides its exploration with a Monte Carlo Tree Search (MCTS) executed at every move. It was the first algorithm to surpass human experts in games such as Go, Chess, and Shogi. AlphaZero therefore marks the first artificial system to consistently outperform human champions in complex and strategic domains without relying on any domain-specific knowledge. It took 4 hours of training on 5,000 first-generation Tensor Processing Units (TPUs) to achieve this [29].

Asynchronous Advantage Actor-Critic

Another important milestone was the development of the Asynchronous Advantage Actor-Critic Algorithm (A3C) framework 1 year before the release of AlphaZero. Actor-Critic methods are a function-approximation-based realization of GPI in DRL. Advantage is an approach to further reduce the variance during training by subtracting the value estimate from the quality estimate (it makes sense to quantify how much better an action is compared to the average action in that state). Further improvements in training stability can be achieved by using asynchronous updates. This is realized by deploying the same environment multiple times in parallel, each interacting with its own agent. Through asynchronous batch updates across the experience of all environments (thus reducing non-stationarity and decoupling updates), it was shown that on-policy algorithms can also be used to learn from experience. The related algorithm is shown in algorithm 2.3 [30].

Algorithm 2.3: Asynchronous one-step Q-learning: pseudo-code for each actor-learner thread based on [30]

Input: Assume global shared θ, θ^-, and counter $T = 0$
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $q(s, a; \theta)$
  Receive new state s' and reward r
  $y \leftarrow$ **if** *terminal* s' **then**
    r
  else
    $r + \gamma \max_{a'} q(s', a'; \theta^-)$
  end
  Accumulate gradients w.r.t. θ: $d\theta \leftarrow d\theta + \frac{\partial (y - q(s,a;\theta))^2}{\partial \theta}$
  $s \leftarrow s'$
  $T \leftarrow T + 1$, $t \leftarrow t + 1$
  if $T \bmod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end
  if $t \bmod I_{async\_update} == 0$ ***or*** *s is terminal* **then**
    Perform asynchronous update of θ using $d\theta$
    Clear gradients $d\theta \leftarrow 0$
  end
until $T > T_{max}$

Trust Region Policy Optimization and Proximal Policy Optimization

Trust Region Policy Optimization Algorithm (TRPO) improves the stability of policy improvements by constraining how much it can change in each update [31].

$$L^{\text{TRPO}}(\theta) = \mathbb{E}_{(s,a)\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}\hat{A}(s,a)\right] \tag{2.12}$$

For formula 2.12, θ are the current policy parameters to be updated. $\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}$ describes the probability ratio between the old policy and the new policy and is multiplied by the estimated advantage $\hat{A}(s,a)$. The loss/surrogate objective function $L^{\text{TRPO}}(\theta)$ in this case is meant to be maximized; high losses suggest that the new policy assigns higher probabilities to good actions (and vice versa) [31].

$$\mathbb{E}_{s\sim\pi_{\text{old}}}\left[D_{\text{KL}}\left(\pi_{\text{old}}(\cdot|s) \,\|\, \pi_\theta(\cdot|s)\right)\right] \leq \delta \tag{2.13}$$

For formula 2.13, D_{KL} is the Kullback-Leibler divergence, measuring the difference between two probability distributions (in this case, the two policies to be compared) and constraining them to be smaller than or equal to a defined δ. More specifically, formula 2.12 is maximized with respect to formula 2.13 [31].

While TRPO is theoretically solid, it is computationally infeasible in practice due to its requirements for second-order optimization [31].

Proximal Policy Optimization Algorithm (PPO) is a simplification of TRPO by keeping the core idea of TRPO (which is to limit big policy updates) and enforcing the limited policy update through a first-order algorithm:

$$L^{\text{PPO}}(\theta) = \mathbb{E}_{(s,a)\sim\pi_{\text{old}}}\left[\min\left(r(\theta)\hat{A}_t,\, \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{2.14}$$

In formula 2.14, $r(\theta)$ defines the policy ratio. By clipping the probability ratio to the interval $[1-\epsilon, 1+\epsilon]$, the update is bounded by ϵ. While PPO lacks the theoretical guarantees of TRPO and therefore should be highly sensitive to the values of ϵ and the learning rate, in practice, PPO works very effectively and efficiently [31].

Relation to the JSSP

RL is often argued to be a natural choice for solving the JSSP and its extensions, such as the dynamic JSSP and the flexible JSSP. This is because RL algorithms can address multiple facets of the challenges involved in creating schedules.
First and foremost, the JSSP can be solved through a partial-based solution algorithm (see section 2.3.4), as RL is a *sequential decision-making process* [13].
Furthermore, the *stochastic behavior* of real Shopfloors can be handled naturally by RL, which is able to adjust its behavior in real-time [32].
Also, RL methods can *discover scheduling strategies* that are superior to human designed

heuristics such as DPR [33].
Multi-Objective optimizations are, in some settings, a necessity in order to align the optimization goals with the defined Key Performance Indicators (KPI). RL is able to *optimize towards composite rewards* and even mix different types of rewards at different stages of training if crafted carefully [32].
Regarding generalization, RL algorithms have been shown to *scale well to unknown instances* if the behavior learned is also applicable to the larger instances, which is a key aspect when scheduling in real production environments [34].
RL algorithms can also be *integrated with real-time feedback* available in Smart Manufacturing Environments, leveraging this feedback for a more optimized scheduling solution [35].
With the ability to use offline learning or online learning, two important aspects of scheduling can be covered by RL: *pretraining* a model based on simulations (offline learning) and *adjusting the model* based on the behavior/feedback during execution (online learning). It remains problem specific if it is desired to learn from past experience in controlled intervals or learn while scheduling [35].

Limitations of RL

While being scalable in theory, the *computational costs* of RL algorithms can be very high, especially when combined with neural networks due to the specialized hardware necessary. This is also related to the high volume of training data needed, which can be expensive to create in the first place, depending on the environment. In essence, even RL algorithms do not fully overcome the curse of dimensionality [1].
Additionally, the *tradeoff between exploration and exploitation* mentioned in section 2.4.4 remains an issue in order to effectively escape local optima [1].
Furthermore, the *design of the reward function* is key to convergence, as it must describe what needs to be achieved, not how. This may be straightforward for, e.g., board games such as chess, where failure/success is solely defined by who wins the game. For more complex problems such as the JSSP, where potentially multiple objects are relevant, it may not always be clear how the reward should be exactly defined [1].
As with most of the AI techniques available, *explanations* of why a certain action was selected are *not very intuitive* (if they can be provided at all), often decreasing acceptance in real manufacturing environments [1].
RL algorithms can, like most AI learning techniques, *forget important information* over time. Relevant information learned in the first few episodes may be discarded by the algorithm in later episodes. Parts of this so-called catastrophic forgetting can be mitigated with effective sampling techniques [1]
For non-stationary problems, theoretical guarantees for convergence are often hard to prove—i.e., it cannot be guaranteed beforehand that the algorithm will eventually converge [1].

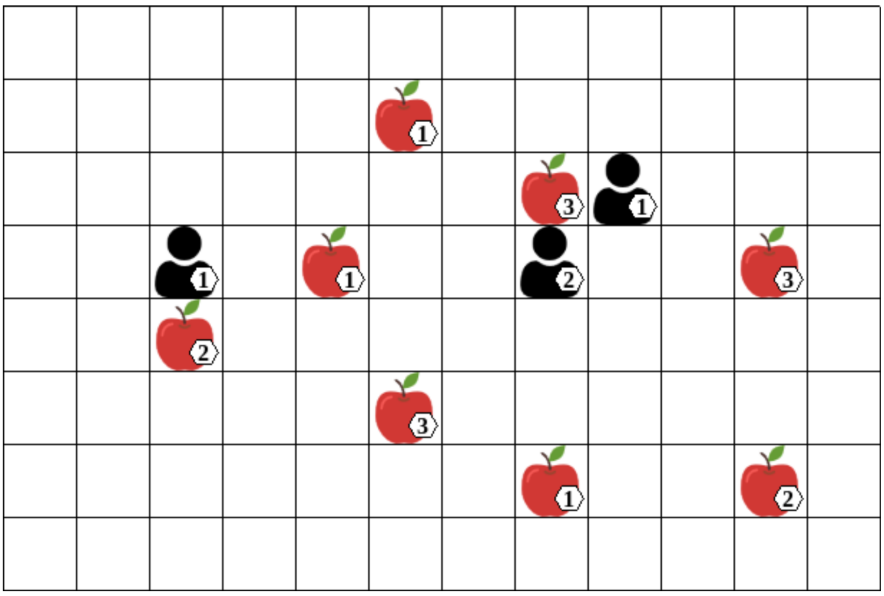


Figure 2.7: A level-based foraging task with three robots [28]

2.4.9 Multi-Agent Reinforcement Learning

Most applications of RL, such as robotics and autonomous driving, naturally fall into the realm of MARL. With the latest AI-Boom and improved successes in single-agent RL, a lot of literature has arisen on the domain of MARL. Though empirically successful, it often lacks a theoretical foundation and is only evaluated empirically [11].

MARL also addresses sequential decision-making processes, but with more than one agent involved. Both the evaluation of the state and the reward received by each agent are influenced by the joint actions of all agents. Each agent has its own long-term reward to optimize, which now becomes a function of the policies of all other agents [28].

Figure 2.7 shows a simple example of a MARL system. The 3 black icons are robots/agents that have the goal of picking up all the apples. The number indicates the 'strength' of the apples and the agents. One or more agents may pick up an apple; the strength of the agent(s) must be equal to or greater than the strength of the apple. An agent is defined to have 6 actions at a given time step: *up*, *down*, *left*, *right*, *collect*, and *no-operation*. agents may 'see' the whole grid or only a part of it [28]. Note how even this simple example would be significantly more challenging to learn with only one agent. One agent would need to learn all combinations of states of each robot. For three robots, this would be $6^3 = 216$ states. This highlights the advantage of MARL compared to single-agent RL in many practical applications.

Markov Games

A Markov Game (MG) is defined by a tuple $(\mathcal{N}, \mathcal{S}, \{\mathcal{A}^i\}_{i\in\mathcal{N}}, \mathcal{P}, \{\mathcal{R}^i\}_{i\in\mathcal{N}}, \gamma)$, where $\mathcal{N} = \{1, ..., N\}$ denotes the set of $\mathcal{N} > 1$ agents, $\mathcal{S}$ denotes the state space observed by all agents, and A^i denotes the action space of agent i. Let $\mathcal{A} := \mathcal{A}^1 \times ... \times \mathcal{A}^N$; then $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ denotes the transition probability from any state $s \in \mathcal{S}$ to any state $s^{'} \in \mathcal{S}$ for any joint action $a \in \mathcal{A}$; $\mathcal{R}^i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function that determines the immediate reward received by agent i for a transition from (s, a) to $s^{'}$; $\gamma \in [0, 1)$ is the discount factor [11].

At time t, each agent $i \in \mathcal{N}$ executes an action a_t^i according to the system state s_t. The system then transitions to state s_{t+1}, and rewards each agent i by $r^i(s_t, a_t, s_{t+1})$. The goal of agent i is to optimize its own long-term reward by finding the policy $\pi^i : \mathcal{S} \to \Delta(\mathcal{A}^i)$ such that $a_t^i \sim \pi^i(\cdot|s_t)$. As a consequence, the value-function $V^i : \mathcal{S} \to \mathbb{R}$ of agent i becomes a function of the joint policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ defined as $\pi(a|s) := \Pi_{i\in\mathcal{N}}\pi^i(a^i|s)$. In particular, for any joint policy π and state $s \in \mathcal{S}$ [11]:

$$V^i_{\pi^i,\pi^{-i}}(s) := \mathbb{E}[\sum_{t\geq 0} \gamma^t r^i(s_t, a_t, s_{t+1})|a_t^i \sim \pi^i(\cdot|s_t), s_0 = s] \tag{2.15}$$

where $-i$ represents the indices of all agents in $\mathcal{N}$ except agent i. Therefore, the solution concept of a MG deviates from that of an MDP, since the optimal performance of each agent is controlled not only by its own policy but also by the choices of all other players in the game. The solution is often described through a Nash Equilibrium (NE). A NE of the MG is a joint policy $\pi^* = (\pi^{1,*}, ..., \pi^{N,*})$, such that for any $s \in \mathcal{S}$ and $i \in \mathcal{N}$ [11]:

$$V^i_{\pi^{i,*},\pi^{-i,*}}(s) \geq V^i_{\pi^i,\pi^{-i,*}}(s) \quad \forall\, \pi^i \tag{2.16}$$

A NE characterizes an equilibrium point π^*, from which none of the agents has any incentive to deviate. In other words, for any agent $i \in \mathcal{N}$, the policy $\pi^{i,*}$ is the best response of $\pi^{-i,*}$. As a standard learning goal for MARL, a NE always exists for finite-space infinite-horizon discounted MG, but it may not be unique in general. Most of the MARL algorithms are designed to converge to such an equilibrium point, if it exists [28].

There are 3 different settings for MGs: cooperative, competitive, or a mix of both.

Cooperative settings refer to situations where all agents optimize towards either a common reward or a team-average reward.
For common rewards, all agents receive the same reward at each timestamp, leading to the same value function and Q-Function for each agent and enabling the same type of updates as in single-agent RL. Such models are sometimes referred to as multi-agent MDP [11].
The average-reward model may use different reward functions for different agents, which may also not be shared with other agents. The goal is to maximize the long-term reward according to the average reward:

$$\overline{R}(s, a, s') := N^{-1} \cdot \sum_{i\in\mathcal{N}} r^i(s, a, s') \quad \forall\, (s, a, s') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S} \tag{2.17}$$

While average-reward formulations simplify decentralized MARL, they often require communication protocols to enable coordination. Designing such protocols is highly task-specific, and attempts at generalization confront core challenges in formal linguistics, such as symbol grounding and meaning. Thus, generalized agent communication parallels the unresolved problem of how language conveys meaning [11].

Competitive Settings are often modeled as zero-sum MGs:

$$\sum_{i \in \mathcal{N}} r^i(s, a, s') = 0 \quad \forall\, (a, a, s') \tag{2.18}$$

The zero-sum game leads to a winner and a loser (in a two-agent setting). The NE leads to a robust policy optimizing for the worst-case long-term reward; a prominent example of such a game would be the Prisoner's Dilemma [11].

Mixed Settings, also known as general-sum games, set no restrictions on the goals and relationships of the agents. Mixed settings can also be a nested structure of a competitive and cooperative setting, for example, for a game with 2 teams and 5 players [11].

Extensive-Form Games

MGs are designed for settings where the environment is fully observable. Extensive-Form Games (EFG) handles settings where (at least some agents) only observe part of the environment:

- An EFG is defined by $(\mathcal{N} \cup \{c\}, \mathcal{H}, \mathcal{Z}, A, \{r^i\}_{i \in N}, \tau, \pi^c, \mathcal{S})$
- $N = \{1, \ldots, N\}$ denotes the set of $N > 1$ agents
- c is a special agent called *chance* or *nature*, which has a fixed stochastic policy that specifies the randomness of the environment.
- $\mathcal{A}$ is the set of all possible actions agents can take.
- $\mathcal{H}$ is the set of all possible histories, where each history is a sequence of actions taken from the beginning of the game. $A^i(h) = \{a^i \mid ha^i \in \mathcal{H}\}$ denotes the set of actions available after a non-terminal history h.
- An agent takes action $a \in \mathcal{A}^i(h)$ given history $h \in \mathcal{H}$, leading to a new history $ha^i \in \mathcal{H}$.
- $\mathcal{Z} \subseteq \mathcal{H}$ is the subset of *terminal histories* that represents the completion of a game.
- A utility is assigned to each agent $i \in \mathcal{N}$ at a terminal history, given by the function $r^i : \mathcal{Z} \to \mathbb{R}$.
- $\tau : \mathcal{H} \to \mathcal{N} \cup \{c\}$ is the *identification function*, specifying which agent acts at each history.

- If $\tau(h) = c$, the chance agent takes an action a according to its policy r^c, i.e., $a \sim r^c(\cdot|h)$.
- S is a partition of $\mathcal{H}$ such that for any $s \in \mathcal{S}$ and any $h, h' \in s$, $\tau(h) = \tau(h')$ and $\mathcal{A}(h) = \mathcal{A}(h')$. Thus, h and h' are indistinguishable to the agent about to act. Elements of $\mathcal{S}$ are called *information states*.
- h is a prefix of h' ($h \sqsubseteq h'$) if h' can be reached from h by taking a sequence of actions.

Agents cannot distinguish between histories in the same information set, which reflects the imperfect information of EFG. However, they can perfectly recall the sequence of information states and actions that lead to their current information state. The perfect recall is crucial to enable the existence of polynomial-time algorithms for solving the game [11].

For any agent, let $S^i = \{s \in \mathcal{S} : \tau(s) = i\}$ be the set of information states of agent i. The joint policy is denoted by $\pi = (\pi^1, ..., \pi^N)$, where $\pi^i : S^i \to \Delta(\mathcal{A}(s))$ is the policy of agent i. For any history h and any joint policy π, the *reach probability* of h under π is defined as:

$$\eta_\pi(h) = \Pi_{h':h'a\sqsubseteq h}\pi^{\tau(h')} = \Pi_{i\in\mathcal{N}\cup\{c\}h}\Pi_{h'a\sqsubseteq h,\tau(h')=1}\pi^i(a|I(h')) \tag{2.19}$$

Similarly, the reach probability of an information state s under π is defined as

$$\eta_\pi(s) = \sum_{h\in s}\eta_\pi(h) \tag{2.20}$$

with a utility of agent i therefore defined as

$$\sum_{z\in\mathcal{Z}}\eta_\pi(z)\cdot r^i(z) \tag{2.21}$$

Challenges

Three common challenges arise in the MARL Theory: non-unique learning goals, non-stationarity, and scalability [11].

Non-unique learning goals are often inferred from the unclear nature of the problem being addressed. It is not even clear if the NE is the best performance criterion for MARL algorithms; solution concepts such as a *cyclic equilibrium* may be more applicable [11].

Non-Stationarity is caused by the fact that multiple agents learn (usually) concurrently, leading to a non-stationary problem from the perspective of the individual agent. The concurrent learning also invalidates the Markov Property [11].

Scalability Issues arise when agents need to account for the joint action space, which increases dimensionality. Accounting for the joint action space reduces the issue of non-stationarity but increases the dimensionality. When MARL is combined with NNs ('deep MARL'), scalability issues can be circumvented. However, the theoretical backbone for such algorithms is very small. While deep MARL algorithms have shown success in various domains, the understanding of why they work is rather limited. This is currently an open research field [11].

Available Information Structures

MARL algorithms can be further classified by their information structure.

A centralized setting implies the existence of a central controller, which observes the global state and selects actions for all agents. Robots in a factory may be controlled by a central controller. This leads to simpler coordination, but it is not applicable if the environment is only partially observable [11].

In *decentralized settings with networked agents*, the agents act independently (observations are made locally) but can communicate with each other; information can be exchanged. Autonomous vehicles would naturally fall into this category. This setting includes the challenges imposed by the aforementioned communication protocols [11].

Fully decentralized settings differ in terms of communication; as mentioned in the last paragraph, decisions are made independently, but no communication occurs between the agents. This is a realistic and scalable approach, but learning cooperative behavior is more challenging [11].

Deep MARL

As in single-agent settings, NNs can be used as function approximators in MARL. This introduces the benefits and challenges described in section 2.4.8. In the context of MARL, introducing NNs further complicates the theoretical analysis of MARL systems. Therefore, most MARL approaches that leverage NNs remain mostly empirically validated, with limited theoretical guarantees of convergence [28].

2.5 Graphs

Graph structures emerge naturally in many different domains, such as molecules, social structures, and financial networks.
Graph theory describes a Graph G through an ordered pair $(\mathcal{V}, \mathcal{E})$, where:

- $\mathcal{V}$ is a finite set of **vertices** (often called nodes)

- $\mathcal{E} \subseteq \{\{u, v\} | u, v \in \mathcal{V}, u \neq v\}$ is a set of **edges**, each edge being an unordered or ordered pair of distinct vertices $E \subseteq \mathcal{V} \times \mathcal{V}$

2.5.1 Graph Neural Networks

One can attach vectors to those nodes for representation learning with a GNN: $x_u \in \mathbb{R}^k$ for $u \in \mathcal{V}$. The resulting matrix for a ML model is then of the form $X \in \mathbb{R}^{|\mathcal{V}| \times k}$ with $X = [x_1, x_2, ..., x_{|\mathcal{V}|}]^T$ [36].

Edges are often represented through an adjacency matrix $A \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$:

$$a_{uv} = \begin{cases} 1 & (u, v) \in \mathcal{E} \\ 0 & x \leq 0 \in \mathcal{E} \end{cases} \tag{2.22}$$

Nodes and edges are unordered. Therefore, for any permutation PAP^T with a permutation matrix P, the following rules must be satisfied:

$$\begin{aligned} f(PX, PAP^T) &= f(X, A) \quad \text{(Invariance)} \\ F(PX, PAP^T) &= PF(X, A) \quad \text{(Equivariance)} \end{aligned} \tag{2.23}$$

where f and F return Graph or node-level outputs [36].

A neighborhood $\mathcal{N}_u$ defines the locality where GNNs can operate on a node level:

$$\mathcal{N}_u = \{v \,|\, (u, v) \in \mathcal{E} \quad \vee (v, u) \in \mathcal{E}\} \tag{2.24}$$

Similarly, a multi-set of all neighborhood features $X_{\mathcal{N}_u}$ is defined as:

$$X_{\mathcal{N}_u} = \{\{x_v \,|\, v \in \mathcal{N}_u\}\} \tag{2.25}$$

Defining a function ϕ over a neighborhood results in:

$$h_u = \phi(x_u, X_{\mathcal{N}_u}) \qquad F(X) = [h_1, h_2, ..., h_{|V|}]^T \tag{2.26}$$

How to best define ϕ is one of the most active research areas in ML. Most research classifies GNNs into either Convolutional, Attentional, or Message-passing:

$$h_u = \phi(x_u, \underset{v \in \mathcal{N}_u}{\oplus} c_{vu}\, \psi(x_v)) \qquad \text{(Convolutional)} \tag{2.27}$$

$$h_u = \phi(x_u, \underset{v \in \mathcal{N}_u}{\oplus} a(x_u, x_v)\, \psi(x_v)) \qquad \text{(Attentional)} \tag{2.28}$$

$$h_u = \phi(x_u, \underset{v \in \mathcal{N}_u}{\oplus} \psi(x_u, x_v)) \qquad \text{(Message-passing)} \tag{2.29}$$

Where ϕ and ψ are NNs. $\oplus$ is any aggregator that is permutation-invariant, such as sum, average, or max [36].

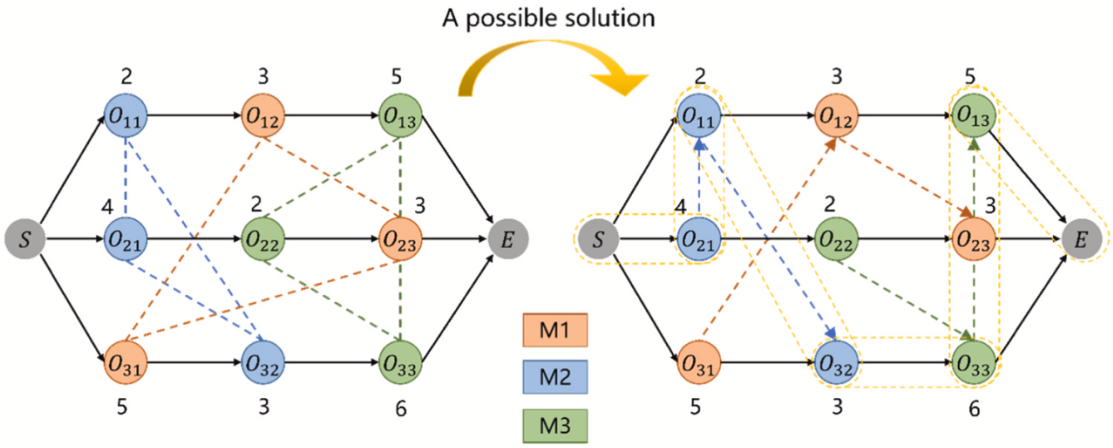


Figure 2.8: JSSP Instance Represented In A Graph [37]

2.5.2 Relation to the JSSP

Graphs can be used as a representation of a JSSP instance. One way to transform a JSSP into a Graph is shown in figure 2.8. There are multiple ways to define this transformation. In the left graph, jobs are defined as nodes, whereas the colored edges define which machine can be used for the execution of the job. The right graph denotes one possible solution for the JSSP; see yellow highlights for the execution order [37]. While this specific JSSP instance is modeled as static, the same logic extends to the dynamic JSSP, as new jobs just need to be inserted into the graph. The flexible JSSP can be modeled with such a graph too, as this just increases the number of edges with the additional constraint that a job can only be processed on one machine.

Regardless of the actual transformation, the Graph structure leveraged for JSSPs is most often disjunctive in the literature. It remains an open discussion in the literature whether disjunctive Graphs are the most effective representation for the JSSP [38, 12].

Using representation vectors on nodes and features, one can learn embeddings for this structure with GNNs. In general, an embedding is a learned function mapping some type of object to a (often dense) real-valued vector of fixed dimensions in continuous space. Such an embedding should contain the semantics of the objects that are useful or necessary for the task. All common types of NNs learn embeddings in different ways. For GNNs, embedding functions aim to capture the Graph structure itself [39].

In the vector space for the GNN embeddings, similar jobs may be closer together, and dissimilar jobs may be pushed apart. This embedding can be trained in a self-supervised fashion and used as a state representation for solving the JSSP [38, 12].

A Graph-based representation of the JSSP often uses most or all of the following Graph properties:

- The Graph has **directed edges** to enforce the sequence for an order.
- the **edges** are **weighted** to capture features such as processing time.
- The Graph is classified as a **heterogeneous Graph**. Graph Heterogeneity describes that there are different types of nodes and edges, such as job and machine nodes.
- For dynamic JSSPs, the **topology** of the Graph **changes** over time as jobs are completed.

2.6 Mapping the Requirements of Smart Manufacturing with MARL and a Graph-Based Representation

This section connects Smart Manufacturing, the JSSP, MARL, and Graphs into one framework. This will be done in two steps:

- Connect Smart Manufacturing and MARL
- Connect MARL and Graphs when solving the JSSP

2.6.1 Connecting Smart Manufacturing and MARL

Figure 2.9 shows a mapping between features of Smart Manufacturing and features of MARL [40].

The following list contains arguments on selected MARL features and why they fit a Smart Manufacturing environment so well:

- **Scalability** fits MARL due to its decentralized decision-making ability [40]
- A **global optimization goal** can be defined by using a global reward function [1, 40]
- The **self-adaptation** and **robustness** are given in an online learning setting [32]
- The need for **decentralized decision-making** fits the decentralized nature of MARL [1, 40]
- Through **Communication** between agents, the need for **negotiation** among certain actors on the Shopfloor can be addressed [28]
- Dealing with **uncertainty** fits the RL architecture in general, as it learns from interactions that are already modeled through a MDP, which is stochastic by definition [28]

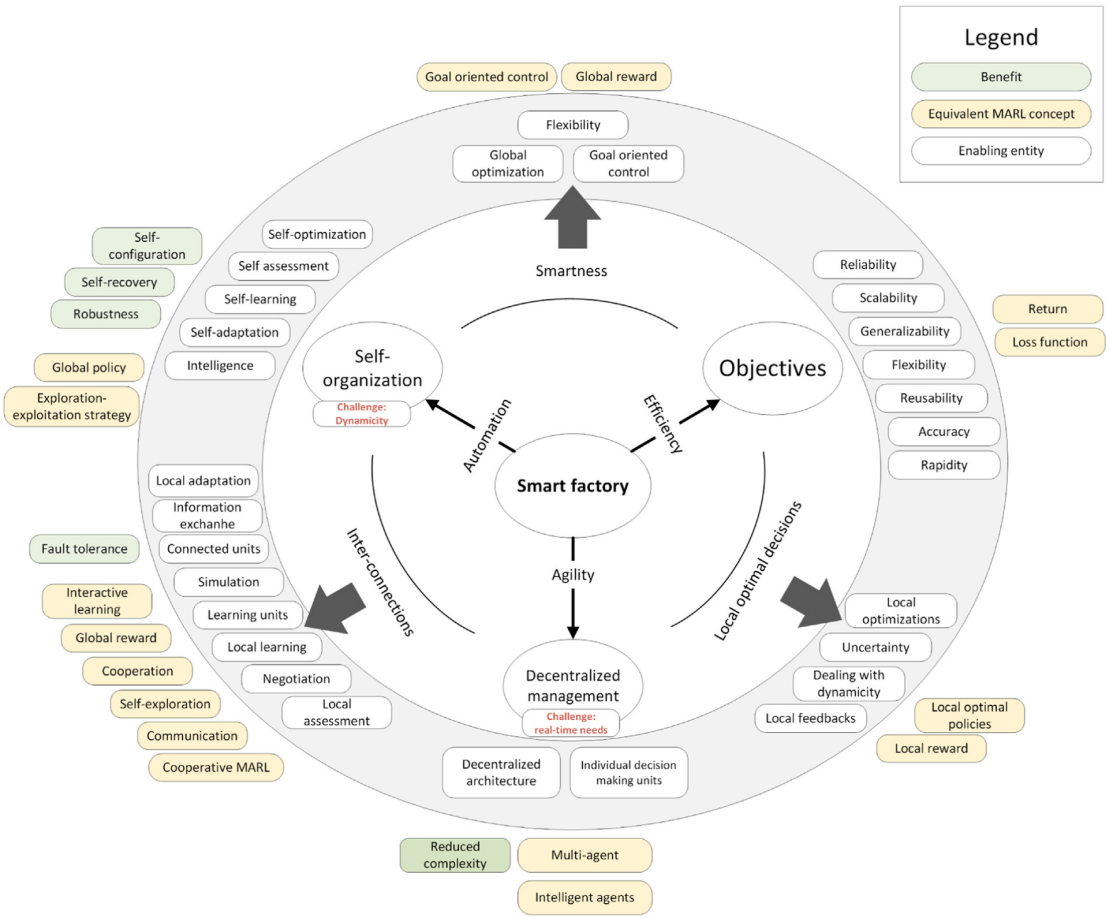


Figure 2.9: Mapping From Smart Manufacturing Features To MARL [40]

- The ability to handle uncertainty and disruptions also enables more resilient and fault-tolerant scheduling [40]
- Through **reusing** existing embeddings and architectures from previous training, MARL algorithms can make decisions **rapidly** and may be **generalizable** [28]

However, all the drawbacks mentioned in section 23 still apply and must be addressed for the specific problem type and size.

2.6.2 Connecting MARL and Graphs When Solving the JSSP

When MARL is used for JSSP instances, the algorithm needs a representation of the JSSP. In general, there are three different ways for this representation [12]:

- Matrix-based

- Statistics-based
- Graph-based

A *matrix-based* state representation uses one or more matrices, one matrix for each feature for jobs and for machines. The dimensionality of the matrices depends on the number of jobs and the number of machines. These raw features may be further processed for an embedding; a CNN would be a natural choice [12].

In *statistics-based* state representations, a set of statistical indicators is used to describe the static and dynamic properties of the jobs and machines. Examples of such indicators are the total number of jobs, total processing time, etc. [12].

Matrix-based state representations have the benefit of being very intuitive and easy to calculate; however, they depend on an exact size definition, which is neither given in smart manufacturing environments nor in the dynamic JSSP. Embeddings created cannot be reused if the problem size changes; this approach is therefore neither scalable nor generalizable [12].

Statistics-based state representations have shown good results in academia and in the industry, but they are mostly handcrafted. Which statistics matter more in a given context and how those statistics should be weighted is highly case-specific and therefore also not generalizable [12].

On the other hand, *Graph-based* state representation can fulfill all of those properties:

- Embeddings can be reused due to the node-level representation
- The size can be adapted during runtime without re-training being required, enabling scalability
- It can generalize under the condition that the learned node and edge representations capture the relevant relationships and constraints of the JSSP domain

Additionally, Graphs have the following benefits:

- Relationships (i.e., edges) are heavily emphasized in Graphs; the relations between orders, jobs and machines can be captured by their nature.
- GNNs are permutation-invariant, and so is the JSSP
- All different kinds of JSSP instances (flexible, dynamic, distributed, etc.) can be modeled through the same architecture

- Due to its easy adaptation for dynamic events, the resilience of the Shopfloor can be increased.

This leads to the conclusion that using Graphs to provide the state representation for MARL is a very reasonable and natural approach when solving the dynamic and flexible JSSP.

CHAPTER 3

Related Work

This chapter describes the related work and the identified research gaps. It is split into 4 sections.
Section 3.1 describes the process of how the works were identified and filtered regarding their relevance.
Section 3.2 describes the commonalities and differences of the identified works. The goal is to get an overview of the State-Of-The-Art (SOTA).
Section 3.3 describes the research gaps that this master's thesis aims to close.
Section 3.4 covers other approaches that are not used in this master's thesis.

3.1 Works Identified

At the beginning, a list of search strings has been set up to identify the first batch of relevant literature:

- Scheduling Reinforcement Learning
- Machine Scheduling Reinforcement Learning
- job-Shop Scheduling Reinforcement Learning
- dynamic job-Shop scheduling
- flexible job-Shop scheduling
- Scheduling Reinforcement Learning Graph
- Smart Manufacturing Multi-Agent
- multi-agent Reinforcement Learning

- Graph Neural Networks
- Scheduling Industry 4.0
- JSSP Graph Neural Networks multi-agent Reinforcement Learning

This versatile list covers search strings for every aspect of this master's thesis. The databases searched were 'IEEE Xplore', 'Springer Link', 'ScienceDirect', and 'Google Scholar'. 31 works have been identified.

Literature from before 2019 was excluded. This exclusion has been done based on the argument that, with the heavily increased computational resources, the new architectures available, and the changed requirements of the industry (see section 2.6), literature from before 2019 is not SOTA anymore. After applying this filter, 21 works remained.

The resulting literature was further filtered based on the similarity to the definitions for the dynamic and flexible JSSP as defined in sections 2.3.2 and 2.3.3. Literature where the algorithms learn to select dispatching rules instead of jobs was excluded[1], as it is not the focus of this thesis. Works that used energy efficiency as an optimization goal were excluded; see the argument in section 3.4. A more detailed explanation, together with an extended argument for this exclusion, is given in section 3.2. After applying these three filters, 12 works remained.

Literature that focuses on other variants, such as the distributed JSSP, was excluded. After applying this filter, seven works remained. Those works can be considered SOTA and relevant to the problem statement defined in this master's thesis.

Based on the works identified, a limited backward search has been performed. Recurring references across all seven works have been identified. The works found during the backward search may be considered fundamental. The goal of the backward search was to:

- Get an overview of the common problem definitions of the JSSP
- Identify common definitions of dynamic events
- Identify common reward signals and how the sparsity of rewards is handled/prevented.
- Understand which types of NNs are used to solve the problem in terms of architectures (such as CNN), depth, and other applied techniques (such as dropout)

Park et al. [13] and Wang et al. [41] have been identified as the most influential works. The five other works identified are all built upon either Park et al. [13] or Wang et al. [41]. A discussion of these seven works is available in section 3.2.

[1] with the exception of Liu et al. [37], as this work provides detailed descriptions of the architecture and will be used for a few comparisons in the next chapter

3.2 Commonalities and Differences of Identified Works

This section covers the commonalities and differences of the seven works identified in section 3.1 and compares their methodological approaches.

While all approaches involve NNs for DRL, **none of the seven works expose details regarding the DRL architectures tested** during the development. An evaluation of different architectures is completely missing. It is therefore not possible to compare the works against each other in terms of their commonalities and differences regarding the NN architecture. It is not clear if the authors of the mentioned works performed such an analysis with respect to the different instance sizes.

Not every work provides details of the exact **definition of job delays and machine breakdowns**; to be precise, it is often unclear if those delays are purely random or at least partially based on the originally expected duration. In the case of machine breakdowns, it is most often unclear how often such breakdowns occur and how the duration of the breakdown is determined. An exception is the work of Wu et al. [42]: this work defines job deviations either by a normal distribution or a uniform distribution. For the instances with a normal distribution, the standard deviation σ is set to 1, 2, and 3, respectively, while the mean μ is set to the expected job duration. For the instances with a uniform distribution, the job duration deviation is bounded by either 10%, 20%, or 30%. For all those combinations resulting from those definitions, either 25%, 50%, 75%, or 100% of the jobs are affected.

It is often argued in the literature that the only **makespan at the end of the episode**[2] **as a reward signal**. Two arguments are commonly referred to in the literature: the credit assignment problem and the exploration inefficiency. The credit assignment problem describes how agents often have trouble figuring out which actions contributed most to a good or bad makespan with such sparse rewards. The exploration inefficiency describes how, without any further guidance from the reward signal, the agents must explore the search space blindly, leading to an inefficient training process.

Approaches with **intermediate rewards** exist. Wang et al. [41] use a combination of machine utilization and makespan as a reward function. Furthermore, Kayhan et al. [1] list 5 objective functions that can be used as intermediate rewards; those objective functions are also listed in section 2.3.6. In contrast, Park et al. [13] use the concept of a normalized makespan, which is provided as an intermediate reward to the algorithm. The makespan of the current policy is normalized by an ϵ-greedy baseline policy, see equation 7.1:

$$\overline{M}(\pi_\theta, \pi_b) = \frac{M(\pi_\theta) - M(\pi_b)}{M(\pi_b)} \tag{3.1}$$

[2] is too sparse as a reward signal. All works mentioned consider the JSSP as an episodic task, where the terminal state is the completion of all jobs

Here, $M(\pi)$ denotes a makespan induced by policy π, π_b denotes the baseline policy in greedy (test) mode, and π_θ denotes the current policy. Therefore, $\overline{M}(\pi_\theta, \pi_b)$ measures the relative scheduling performance of π_θ to π_b and obviously has smaller variance than $M(\pi_\theta)$. Park et al. [13] can be considered the first Graph-based MARL algorithm on the dynamic JSSP; almost all SOTA works refer to the implemented ScheduleNet algorithm of Park et al. [13]. However, it does not consider a flexible JSSP. Both Park et al. [13] and Wang et al. [41] consider team rewards as described in section 2.4.9 and are classified as partial-based solution algorithms (see section 2.3.4).

In contrast, Lv et al. [43] argue that **individual rewards for each agent** are a more promising approach. Providing good individual rewards for each agent can also be argued well from a theoretical point of view: Agents that contribute more to a good Schedule should be rewarded more highly, and vice versa. Furthermore, it uses a whole-based solution algorithm (see section 2.3.4) in the beginning; after a machine breakdown or a delayed job, the Schedule is then repaired through repair strategies (like moving a job to an alternative machine[3]). Those repair strategies are learned during training.

The most common **evaluation criterion** for the successful learning of a RL algorithm for a JSSP instance is convergence. Convergence is reached if the policy is no longer oscillating but has stabilized, implying that an optimal or near-optimal policy has been found. In the case of the dynamic JSSP, convergence must be considered with regard to the dynamic factors, as a dynamic Shopfloor will have oscillations in the makespan regardless of a good Scheduling algorithm. Therefore, the interval for a RL algorithm considered for convergence on the dynamic JSSP depends on the uncertainty introduced in the given Shopfloor.

There exist different **benchmark datasets** for the static JSSP, such as the benchmarks provided by Taillard [44] and Demirkol [45]. However, there do not exist benchmarks for the dynamic JSSP. The main reason for the absence of such benchmarks is that there is no global definition of the properties of the dynamic JSSP.

Other works use **simple algorithms** (such as FIFO) **as a comparison** against the implemented algorithm. Comparisons against such algorithms do not show that an optimal solution has been found; they only demonstrate that the algorithm itself is better. However, they provide relevant insights if an algorithm is able to learn the desired behavior and showcase effectiveness against current approaches in the industry (see section 2.2.4).

Works such as Liu et al. [37] **evaluate a Schedule based on KPIs**, such as those introduced in section 2.3.6. Such an evaluation helps compare the solutions against each other but provides no additional scientific insight regarding the optimality of the

[3] Lv et al. [43] consider the flexible JSSP

scheduling solution.

There is a fundamental discussion in the domain of Scheduling regarding the **usage of DPRs**: Works such as Wu et al. [42] do not define a job as the selected action of the agents, but as a DPR instead. I.e., the agents' action is a DPR, and the result of applying the rule decides the next job to be executed. A DPR defines a rule-based schema on how to select a job in a given situation. Such a DPR can be, for example, FIFO or Longest-Processing-Time-Remaining (LPTR). The main argument for DPRs is that it is possible to simplify the learning process by introducing a predefined set of rules, reducing the search space of the problem instance to a fixed size and enabling generalization. Another (less scientific) argument is the increased acceptance of such algorithms in the industry, as they are often viewed as less of a black box compared to NNs. In contrast, all other works referred to so far in this section do not use DPRs. The main argument is that the learning algorithms should not have a predefined set of logical schemas, implying that generalization and a predefined schema are not compatible.

Ho et al. [46] analyze the **largest problem instances**. All other works mentioned until now consider instance sizes of a maximum of 15 machines and a maximum of 50 jobs. In contrast, the largest instance of Ho et al. [46] includes 25 machines and 200 jobs. It is also the only work found that states the training happened on a Graphics Processing Units (GPU).

The **architectures of the Graph-based MARL algorithm** in the aforementioned works also share some commonalities; the differences are often induced by a different type of problem statement.
Park et al. [13] use a disjunct graph for the GNN, creating an attentional neighborhood as defined by equation 2.28. Those job embeddings are then used to calculate similarity scores to the machine features. Park et al. [13] do not disclose the features used. Wu et al. [42], Lv et al. [43], and Wang et al. [41] all use a similar approach.
Lv et al. [43] list the features used for jobs and machines. The machine-Features are:

- Machine utilization
- Number of candidate jobs
- Total number of subsequent jobs
- Number of delayed subsequent jobs on the machine.
- Total completion time of all subsequent jobs on the machine.
- Sum of delay times of all subsequent jobs on the machine.
- Maximum delay time of all subsequent jobs on the machine.
- Average delay time of all subsequent jobs on the machine.

- Ratio of delayed jobs
- Sum of delay times of the finished jobs on the machine.
- Completion time of the last finished jobs on the machine.

Different architectures exist. Liu et al. [37] use Multi-Head Mix Attention on the job embedding, combined with a Feed-Forward Neural Network (FFN), resulting in job embeddings with 256 dimensions. Those embeddings are then used for the PPO-based RL agent. Note that the action space of Liu et al. [37] consists of different DPRs. It is therefore not directly comparable with the architecture originally proposed by Park et al. [13].

The **hyperparameters used** are mentioned in some of the works. The works that expose the used hyperparameters all use similar values. For Wang et al. [41], the learning rate is 10^{-4} and the discount factor γ is 0.99, which is consistent across all works. Wang et al. [41] also use a PPO-based algorithm with a clip range of 0.1, which is also consistent with all works that use PPO.

3.3 Research Gaps Addressed by This Thesis

Two relevant gaps in the literature are analyzed in this master's thesis:

- The impact different NN architectures have on the learning of the flexible JSSP for the RL agents when using Graph-based MARL as the scheduling algorithm (RQ1)
- What types of dynamic factors can a Graph-based MARL Scheduling algorithm learn, and what are their differences in learning? (RQ2)

The set of machine features used is described in section 4.2.3 and is similar to the features described in section 3.2. The ideal feature selection in the dynamic JSSP may be an interesting research question open for future work.

This master's thesis considers the dynamic and flexible JSSP; however, most works only focus either on the dynamic JSSP or the flexible JSSP. While the algorithm can be the same if one extends the dynamic JSSP to also be flexible, the solving time drastically increases due to the increased search space. For a flexible JSSP, the search space grows exponentially with respect to the number of jobs. With respect to the complexities mentioned in sections 2.3.2 and 2.3.3, this problem statement is known as one of the most challenging real-world scheduling problems [47, 48].
Delays will be calculated with respect to the originally expected duration of the job. Furthermore, machine breakdowns will have a stochastic repair period defined.
This master's thesis will not include any predefined logic, such as DPR, in the learning process.

As mentioned in section 1.2, the overall architectural design will be similar to Park et al. [13]. A GNN yields job embeddings for currently executable jobs for a given machine. Through a NN, similarity scores are calculated. The job with the highest similarity score is selected as the next job for the given machine.

3.4 Different Approaches Not Covered in This Thesis

To solve the dynamic and flexible JSSP, other approaches exist. This section covers other approaches commonly referred to in the literature.

As already stated, Liu et al. [37] use a different architecture and select DPRs instead of jobs directly. This approach could be classified as a Meta-heuristic algorithm according to figure 2.5. In general, any heuristic or meta-heuristic could be used on the JSSP. However, the focus of this master's thesis is on algorithms with zero previous knowledge.

The number of different types of GNNs that could be used is enormous. The field of GNNs (and AI in general) is currently facing a rapid transformation; new methods and approaches are published regularly. It remains to be discussed which GNN types are the most reasonable on the JSSP from a theoretical point of view.

For MARL, other variations exist, too. Zhou et al. [32] introduce negotiations between machines through communication. Lv et al. [43] provide individual rewards instead of team rewards. Both of these approaches are highly interesting and may be open to future work with the same research questions mentioned in section 1.2.

In this master's thesis, energy consumption is not considered. However, there exist works that solve the energy-aware JSSP. In essence, the makespan and energy usage are combined in a composite objective function, changing the focus of the problem to be solved entirely. In this energy-aware JSSP, the argument that any optimal algorithm must minimize the makespan does not hold anymore.

CHAPTER 4

Proposed Architecture

This chapter describes the architecture for solving the Research Questions in section 1.2. The chapter is broken down into the following sections:

- **Shopfloor Environment**: This section provides a general description of the simulation environment that was created to execute and test graph-based MARL algorithms.
- **Algorithm**: This section shows details of the algorithm used in the simulation.

4.1 Shopfloor Architecture

In order to test a Graph-based MARL algorithm, an environment as described in 2.4 is necessary. The requirements for such an environment are as follows:

- Reproducible
- Simulate Machines
- Handle incoming orders (see section 2.3.1)
- Perform jobs based on the agents' actions
- Provide rewards and the next state to the agent
- Introduce stochastic delays and breakdowns based on defined parameters
- Store all information in a database
- Perform job execution in real time or in simulated time

A great effort was put into providing a full-fledged simulation environment to execute the experiments in this master's thesis. The created environment is able to simulate a Shopfloor (see figure 4.1), consisting of the following major elements:

- An arbitrary number of **machines**
- A **MES** similarly as described in section 2.1.4
- Dynamically incoming orders as **external input**
- An **algorithm** providing parts of a Schedule (see section 2.3.4). This element integrates the agents as defined in section 2.4.9 into the environment.

Parts of this implementation were done during the course '194.147: Interdisciplinary Project in Data Science'. The goal during the project was to provide a basic framework that can be used to implement and test Scheduling algorithms of different types. After finalizing this initial setup, the framework has been adapted to handle scheduling for a Graph-based MARL algorithm; see figure 4.1. The following sections provide detailed explanations of the final architecture used for this master's thesis.

Manufacturing Execution System

As visualized in figure 4.1, the central piece of the simulation is the MES. The MES is responsible for handling external input, communicating with the algorithm and the machines, and storing all the data inside a SQLite database.
The author of this master's thesis acknowledges that this element may be viewed as a mixture between a MES and an APS System. As monitoring and planning are both needed in the simulation, it was decided to concatenate those two functionalities into one element for architectural simplicity, which is ultimately more closely related to an MES than an APS System.

Probabilistic State Machine

Through a config-file, a predefined number of machines with individual configurations can be deployed per Shopfloor. The machine receives job assignments and responds with feedback. The implemented machine for this master's thesis is a probabilistic state machine (see figure 4.2). It is an extension of a finite and event-driven state machine [16].

The probabilistic behavior of the machine is modeled through the parameters α, β, and γ.

- **With probability** α, the job-duration is shifted by a normal distribution (mean and standard deviation can be defined individually in the config-file).
- **With probability** β, a machine is broken and has to be fixed through maintenance. Until this maintenance has occurred, the machine cannot be used.

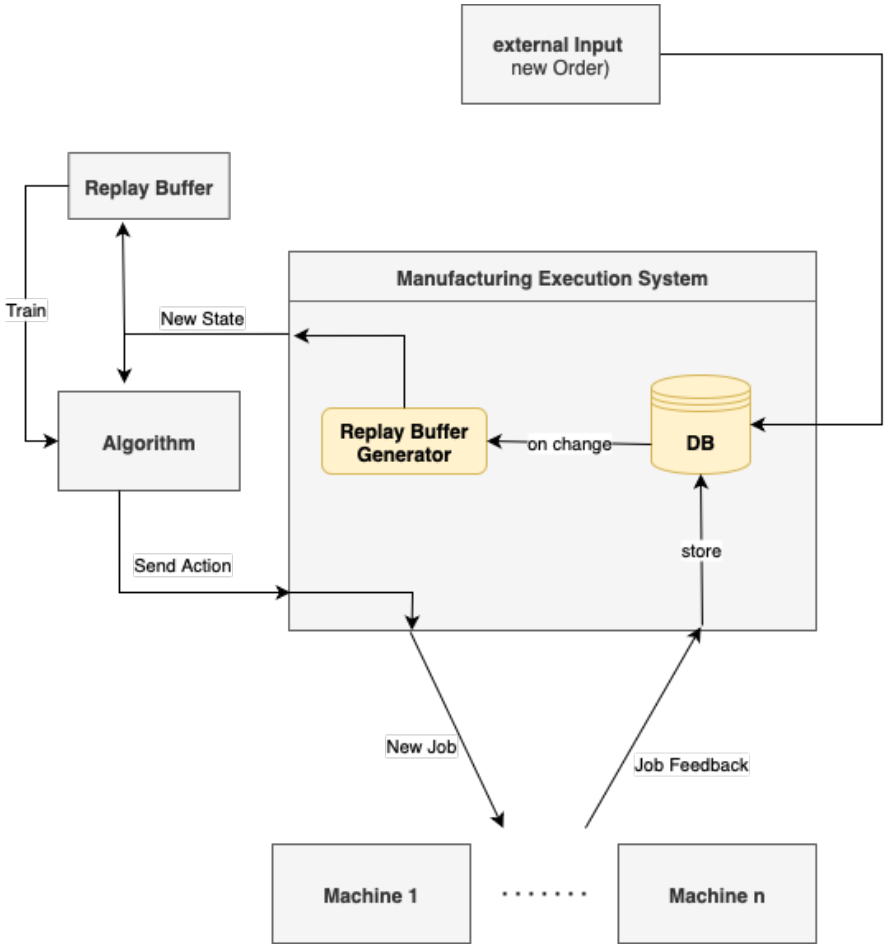


Figure 4.1: Architecture Of The Shopfloor-Simulator

- **With probability** γ, a job is aborted and must be performed again
- **With probability** $1 - \alpha - \beta - \gamma$, the machine works as expected, the job is performed on time.

The states 'Perform Task', 'Abort Task', and 'Machine Fixed' are temporary states. These states are not actually implemented but are added to figure 4.2 because the author considers the improved readability and interpretability of the state machine to be relevant enough to partially violate the definitions in the literature.

During the initialization of the environment, machines have the state *Offline* and go to *IDLE* after the initialization is finalized.
When *Idle*, machines can *PERFORM* a *TASK*. According to the defined probability parameters as described in the last enumeration, 4 outcomes are possible.
If a *TASK* is *COMPLETE* (on time or deviated), the state is set back to *IDLE*. This is

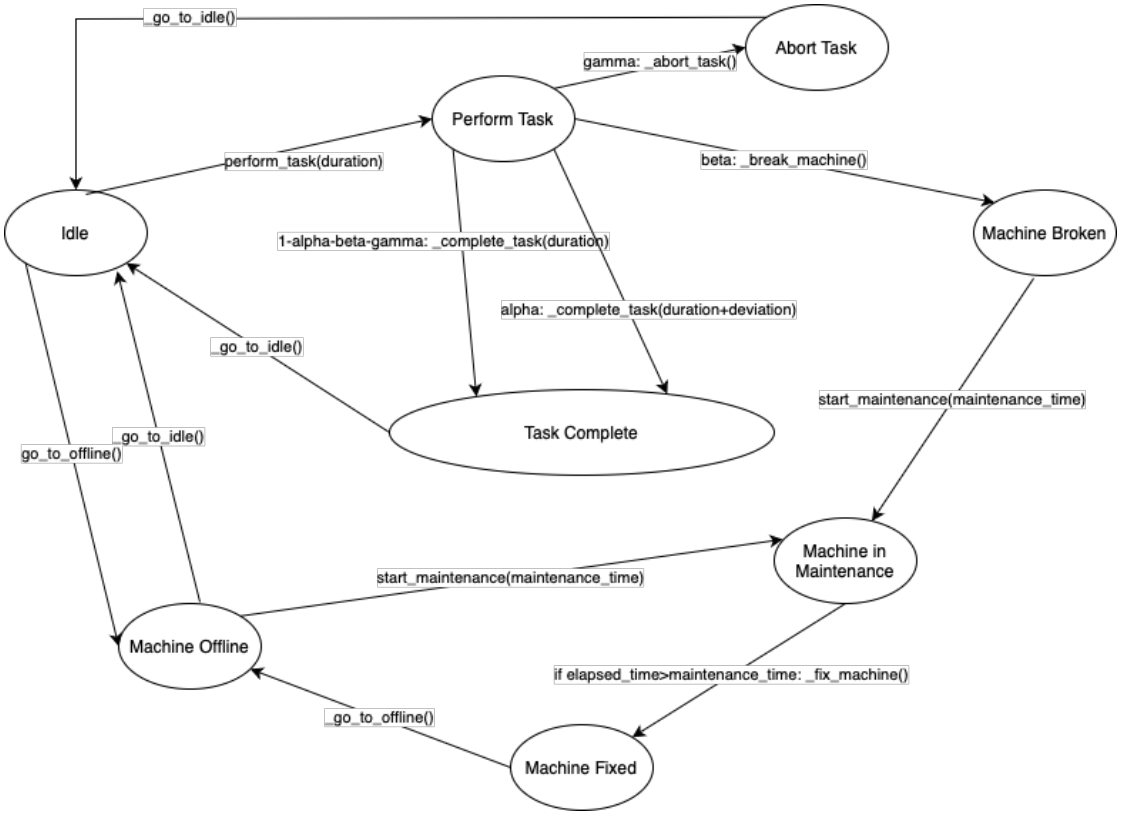


Figure 4.2: Probabilistic state machine

also the same if the *TASK* was *ABORTED*.
If a *MACHINE* is *BROKEN*, a mechanism can activate the state *MACHINE IN MAINTENANCE* to start the maintenance.
If maintenance has ended, then the *MACHINE* is *FIXED* and goes back to the state *MACHINE OFFLINE*, which can then transition back to the state *IDLE*.

Database

The Database is implemented according to the schema in figure 4.3.

- One **order** consists of one or more *jobs*. A **order** must have an *income_date* and a *due_date* set. The fields *finish_date*, *was_canceled* and *cancellation_time* were not used in this master thesis, see section 7.4.3.
- One **job** may have one *predecessor*, which can be used to enforce a sequence (fully or partially) of job execution. A **job** can be executed on every **machine** listed in *appropriate_machines*. The *machining_time* describes the expected duration for execution on any machine.
  The fields *efficiency_factor*, *use_nd_deviation*, *nd_mean* and *nd_variance* are the

parameters introducing a probabilistic behavior as described in section 4.1. The *efficiency_factor* is a constant factor in the interval (0,1) by which is the set job duration is multiplied, while *nd_mean* and *nd_variance* are the parameters to influence the job duration by a normal distribution[1]

- The **machine** relation is versioned through timestamps and contains historical information about the machine during runtime. If the **job** is performed or aborted, or the machine is broken, or if maintenance is performed, a new entry is created in **machine**. The entries provide historical information about the machines during runtime of the simulation; this versioned table further guarantees that all information created during runtime can be stored and the exact sequence of Events/Actions can be reconstructed.
  For versioning, the fields *status_from* and *status_to* are used. The field *available* coincides with the states of the state machine described in figure 4.2. If the state machine is not *Idle* or *machine Offline*, then *available* is False. The fields *alpha*, *beta* and *gamma* are the probabilistic parameters for the state machine as defined in section 4.1.
  The field *seed* is stored redundantly in the Database as an additional verification which seed was used to create the probabilistic behavior.
  The fields *use_nd_deviation_for_duration*, *nd_duration_mean*, *nd_duration_variance*, introduce a probabilistic behavior. Again, the latter two parameters influence the job duration by a normal distribution, while the first parameter defines if a delay should be introduced for a specific instance.

- **Maintenance** must always be mapped to exactly one **machine**. There are two types of maintenance: planned and unplanned.
  If a **machine** breaks according to the parameter β (unplanned maintenance), the **Maintenance** starts immediately and is determined by the defined probabilistic parameters in **machine**.
  On the other hand, external Input can set a time interval where the **machine** is being maintained and not operable (planned maintenance). Maintenance intervals are deemed out of scope for this work and will not will be further discussed.

- **machine Feedback** is the mapping between the machine states from figure 4.2 and a numerical value.

- **job Assignment** can be viewed as the most important Table in this Entity-Relationship-Diagram (ERD). It creates a relationship between **job** and **machine**. Additionally, to incorporate for the case where a job is assigned more than one time towards a machine in case of rescheduling the job (introduced by the probabilistic behavior described in section 4.1), a timestamp (*time_created* is used as an additional unique identifier.
  The field *machine_feedback* is set based on the **machine Feedback** determined

[1] In this work, nd_mean is set to the expected job duration

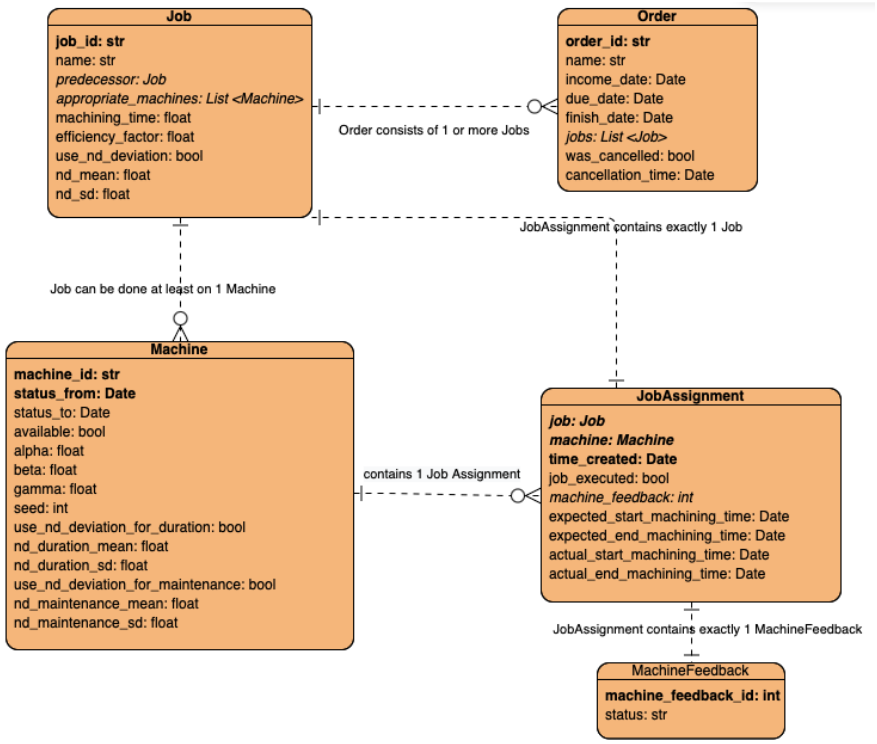


Figure 4.3: ERD of the Database

by the machine during runtime. Dependent on that outcome, the boolean field *job_executed* is set.
The field *expected_machining_time_start* is equal to *time_created*, the field *expected_machining_time_end* is calculated as *expected_machining_time_start* + the *machining_time* from the **job**.
The fields *actual_machining_time_start* and *actual_machining_time_end* are set based on the MQTT messages from the machines during runtime (which contain the state of the state machine and the necessary timestamps).

Modeling of (Probabilistic) Delays

Different types of probabilistic behavior can be modeled. The time actually needed to execute the job on the machine may be called *adjusted_time*.

- **Deterministic job Delays**: One can model jobs with a consistently higher (or lower) *machining_time* through the *efficiency_factor*. The adjusted time is calculated as:
  $adjusted\_time = \frac{machining\_time}{efficiency\_factor}$.

- **Probabilistic job Delays** On the other hand, one can alter the *machining_time* by adding a delay sampled from a given normal distribution. The adjusted time is calculated as:
  $adjusted\_time = machining\_time + \epsilon, \quad \epsilon \sim \mathcal{N}(m, \sigma)$, where m equals a prefined value sampled during order creation.

- **Probabilistic Machine Delays** With probability α, the *machining_time* is altered by adding a delay sampled from a given normal distribution. The adjusted time is calculated as:
  $adjusted\_time = \mathcal{N}(machining\_time, \sigma)$

- **Probabilistic Machine Breakdown** With probability β, the machine has a breakdown and needs to be maintained. The maintenance time is a constant factor adjusted by a delay sampled from a given normal distribution.

Delays for machines and jobs are supported via an *efficiency_factor* in the interval (0,1] and a normal distribution, both configurable independently. This allows, for example, to simulate consistently slower machines to test for potential adaptation in the algorithm.

Although all delay types can be combined, it is assumed that algorithms can only learn one or two types of delays at a time.

Docker

The simulation is split into different Docker containers inside one Docker network. The following containers are being deployed:

- One Container for **each machine**
- One Container for **the MES**
- One Container for **the algorithm**

Additionally, through Docker Volumes, the Database and the replay buffer are persistent. The Container containing the algorithm is also connected to this directory.

MQTT Communication

For communication between the Docker containers, the MQTT protocol has been used. The MQTT protocol provides lightweight and fast communication between the Docker containers while ensuring that messages are delivered exactly once. This promise can be achieved by setting the Quality of Service to level 2 [49].

Discrete Event Simulation

The MES contains an Event-Queue, where each incoming MQTT Message is treated as a single Event. Each MQTT Message is sent with a timestamp; the Queue is automatically sorted by this timestamp after each Event is added. To guarantee the correct sequence regarding the simulation steps, a delay parameter of 0.5 seconds is set by default to counteract potentially delayed communication between the containers.

Three different modalities are implemented: real time, adjusted real time, and simulation time.
In *real time*, the job durations need as many seconds as defined (shifted by the stochastic behavior).
Adjusted real time, the real time is adjusted by a constant factor.
In *simulation time*, the durations are reported immediately by the machines, leading to an accelerated simulation, with the MES reaction time as the main bottleneck as all messages must passt the MES.

Replay Buffer Generator

For any relevant change[2] in the Database, the replay buffer Generator is triggered. The replay buffer is created according to the definition provided in section 2.4.7.

The generator creates the new state by accessing the database. Afterward, it writes into a file containing the replay buffer while also sending the new state to the algorithm.

The replay buffer Generator is both sending the new state to the algorithm and creating the replay buffer. If the algorithm had created the replay buffer, the implementation of re-using replay buffers would have been more complicated.

Algorithm

As described in section 4.2, the algorithm reacts to a message provided by the MES according to the following rules:

- If a job was either **successfully performed or aborted** on a given machine, pass the corresponding agent the new state. The agent must **select a new job**.
- If the **machine is broken**, **do not** let the agent **select a job** until maintenance has ended on that machine.
- If **maintenance has ended** for a given machine, make the agent **select a new job**.

[2]Relevance with regard to the creation of the replay buffer

- **If** an agent selected a **new job**, send a corresponding **MQTT message** to the MES.

Parallel Simulations

Multiple simulations can be deployed in parallel to increase disk and memory utilization in the Exoscale cluster. As simulations may need to be (re)started independently, the image names and container names are guaranteed to be unique to ensure that the shutdown or replacement of one simulation does not affect the other.

Logging

The simulation automatically logs the output of the entire simulation at different levels. For each simulation, a unique folder is filled with:

- The output of **each container**
- The **Database**
- The **replay buffer**
- the **configurations and artifacts** created by the library 'Weights & Biases' (wandb). Those files created by each run are additionally uploaded automatically to the wand project-dashboard for a more thorough investigation of the outcomes [50].

Order Generation

An order Generator has been defined to create orders with a defined set of constraints. The parameters for creating a list of orders are as follows:

- **number_orders (int)**: number of orders
- **number_jobs (int)**: number of jobs per order
- **use_predecessor (bool)**: if True, each job inside an order can only start once the previous job is finished (with the exception of the first job)
- **num_machines (int)**: the number of machines
- **flexible (int)**: a value in the interval $[0, 1]$. If flexible=0, then each job can only be executed on one randomly sampled machine. If flexible=1, each job can be executed on every machine (full flexibility according to chapter 2.3.3). If the value is between 0 and 1, it is used to sample a given fraction of machines where the job can be executed. It is guaranteed by default that each machine is used by at least one job over all orders generated through one order Generation.

- **job_delay_ratio (float)**: this value defines the fraction of jobs that will have a delay in the simulation later on.
- **sample_exact_number_jobs (bool)**: if False, then the parameter number_jobs per order is randomly sampled with a maximum deviation of 50%. This is a relevant parameter during the training process to increase the variance in the orders shown to the algorithm[3].

Global Clock Management

It is necessary to manage clocks across all parts of the simulation. To provide this functionality, all events across the simulation include a timestamp that relates to the current global time of the simulation. The global clock is managed by the MES.

4.1.1 Experiments / Instance Definitions

In this thesis, an experiment is defined as a set of instances used to train the algorithm. An instance is defined according to 2.3.1. On the Exoscale cluster, a Postgres-Database was created according to figure 4.4 to track results across all experiments.

Postgres Database

The table **experiments** contain the following attributes:

- **experiment_id**: unique ID generated by a hash value from the used yaml file.
- **run_id**: If an experiment was run multiple times (in case of unexpected errors, interruptions, or similar), this continuous id ensured that those runs could be separated.
- **experiment_yaml**: For easier deployment, an entire experiment was defined through a yaml file.
- **created_timestamp**: timestamp at the creation of the experiment
- **execution_on_node**: different nodes on the Exoscale cluster have been used in parallel.
- **execution_start**: timestamp at the start of the first running instance.
- **execution_end**: timestamp at the end of the last finished instance.
- **execution_duration**: the time between execution_end and execution_start in seconds.

[3] During the training process, this parameter was crucial to avoid overfitting

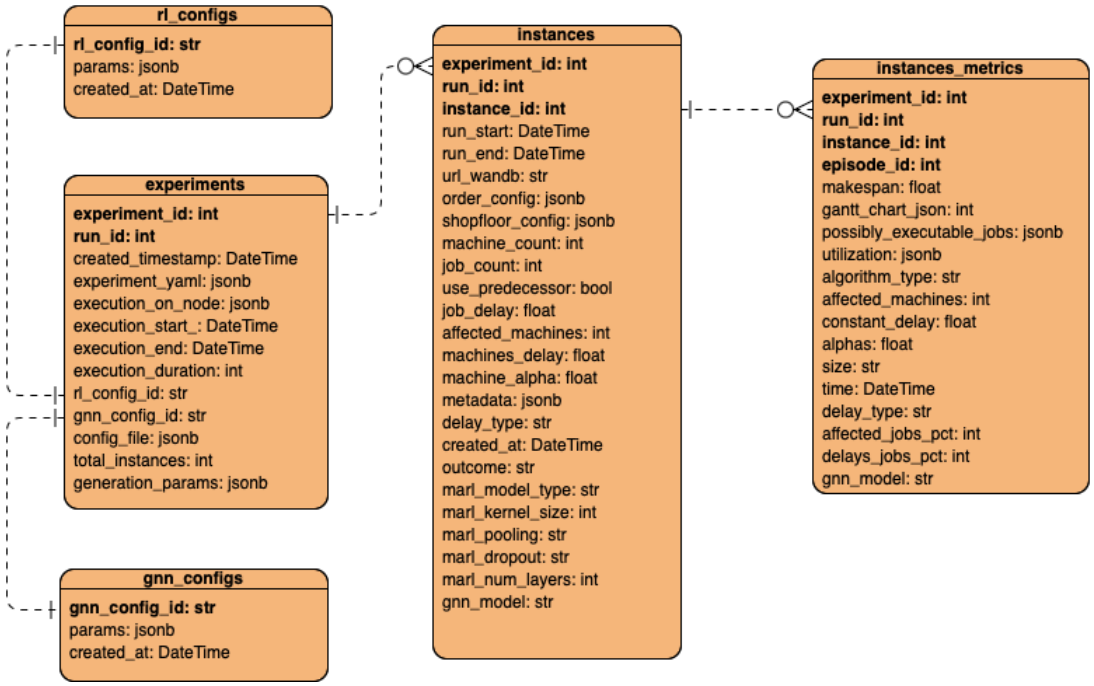


Figure 4.4: ERD of the Postgres DB in Exoscale

- **rf_config_id**: foreign key to the table rl_configs, which contains the specific RL parameters used.
- **gnn_config_id**: foreign key to the table gnn_configs, which contains the specific GNN parameters used.
- **config_file**: the replica of the config file uploaded into Weights & Biases, which is a different variation of experiment_yaml
- **total_instances**: the number of instances deployed through this experiment

The table **instances** has a 1:n relationship to the table **experiments** and contains the following attributes:

- **experiment_id**: foreign key to the table experiments.
- **run_id**: foreign key to the run id of the experiment.
- **instance_id**: unique id, which is derived from the current timestamp of creation and a hash value of the experiment id.
- **run_start**: start of the instance run.
- **run_end**: end of the instance run.

- **url_wandb**: url to the run in Weights & Biases for a more thorough analysis of the instance outcomes.
- **order_config**: the orders used for the simulation.
- **shopfloor_config**: The exact config file of the Shopfloor used, for example, the number of machines, their set deviations, and the classes used inside the simulation (as the simulator was originally designed as a general-purpose Shopfloor simulator)[4] statements.
- **machine_count**: number of machines.
- **job_count**: total number of jobs over all orders used.
- **use_predecessor**: True if predecessors were used as a constraint for the jobs.
- **job_delay**: job delays are defined in the order Generator.
- **affected_machines**: if machine_delays was tested, this attribute describes the number of affected machines by a delay.
- **machines_delay**: the machines delay defined if machines_delay were tested.
- **machine_alpha**: the alpha parameters set for the affected machines according to figure 4.2.
- **metadata**: further metadata relevant during runtime.
- **delay_type**: either 'machine_delays' or 'job_delays'.
- **created_at**: timestamp during the creation of the instance.
- **marl_model_type**: either 'MLP' or 'CNN'.
- **marl_kernel_size**: either 3 or 5 if marl_model_type = 'CNN'.
- **marl_kernel_size**: either 'average' or 'max' if marl_model_type = 'CNN'.
- **marl_dropout**: hyperparameter 'dropout' for the agents, either 0.1 or 0.2
- **marl_num_layers**: hyperparameter for the number of layers used for the agents, an integer between 2 and 5
- **gnn_model**: either 'gat', 'chebnet1', or 'chebnet2' [5]

[4]Some of the features inside this config are also stored as separate attributes such that the analysis of the outcomes could be handled by less complex SQL

[5]in this thesis, 'gat' is used

The table **instances_metrics** was used to analyze the outcomes of the experiments. It is the basis for the plots created to analyze the outcomes over all experiments. It's attributes are:

- **experiment_id**: foreign key to the table experiments.
- **run_id**: foreign key to the run id of the experiment.
- **instance_id**: foreign key to the corresponding instance.
- **episode_id**: episode id
- **makespan**: makespan in this episode.
- **gantt_chart_json**: jobs with timestamps and references to the used machine in order to represent the schedule of this episode in a GANTT Chart.
- **possibly_executable_jobs**: A mapping between machines and the total number of executable jobs on that machine. This was used as an additional check that the simulation worked as expected and the same orders were used in instances to compare with.
- **utilization**: a json file containing the utilization per machine in that episode.
- **algorithm_type**: either 'algorithm_static', 'algorithm_dynamic', 'FIFO', 'SPT', or 'random'.
- **affected_machines**: corresponds to 'affected_machines' from the instances-table.
- **constant_delay**: if a constant machine delay was used, this value was set to either 0.8 or 0.9 depending on the instance definition.
- **alphas**: the alpha parameters set for the affected machines according to figure 4.2.
- **size**: a string formatted as '{machines}x{jobs}' according to the instance definition in section 2.3.1.
- **time**: timestamp when the episode was run.
- **delay_type**: either 'machine_delays' or 'job_delays'.
- **affected_jobs_pct**: if delay_type = 'job_delays', this attribute corresponds to the percentage of jobs affected by a delay and is either 25, 50, or 75, depending on the instance.
- **delays_jobs_pct**: if delay_type = 'job_delays', this attribute corresponds to the percentage of delay per job depending on the original duration. It is either 25, 50, or 75, depending on the instance.

- **gnn_model**: either 'gat', 'chebnet1', or 'chebnet2'

In general, this database was de-normalized on purpose so that the analysis could be achieved with easier SQL Statements.

Difference between Postgres Table 'instances_metrics' and Weights & Biases

The outcomes of the experiments have been tracked in the Postgres DB and in Weights & Biases because both were used for different types of granularity.

Weights & Biases is ideal for analyzing the outcomes of one instance due to its capability of automatically creating different types of plots, making it suitable for visual inspection of individual instances.

In contrast, the table in the Postgres DB allowed for an easy comparison of all instances through SQL Statements.

Additional Implementation Details

This section provides an overview of other implementations that are worth mentioning in this thesis.
Monitor: A monitoring class was implemented to track all instances inside a run. This class was used during runtime to track all instances of an experiment, logging the results in Weights & Biases and in the Postgres DB, starting and shutting down instances, and determining the end of an experiment. It was implemented asynchronously to handle all instances in parallel.

4.2 Algorithm

The algorithm can be described in 3 components:

- A rule-based class called the **RL Coordinator**[6]. The RL Coordinator is managing the other two components.
- **Agents** that select jobs.
- A **Graph State Manager**, which keeps a Graph state representation of the JSSP-Instance.

Figure 4.5 shows a schematic overview of how jobs were selected and executed. The overall process can be described in nine steps, which are as follows:

[6]Note that this module is not a DPR as mentioned in section 3.2, but it is necessary for coordination

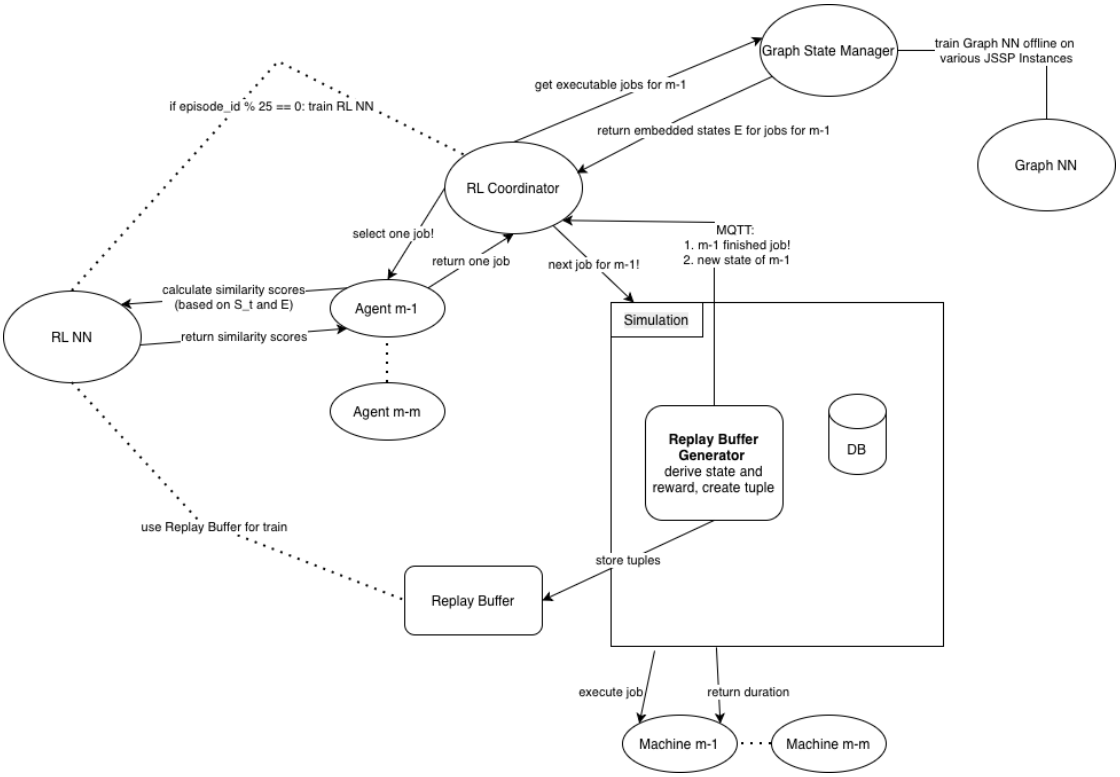


Figure 4.5: job Execution Pipeline

- As soon as an episode starts, the RL Coordinator tries to assign a job for each machine. Each machine has its own agent that keeps track of whether the machine is idle, along with other information (such as the global clock) necessary during runtime.

- The Graph State Manager yields information about which jobs can currently be executed on a given machine and returns those jobs along with their embeddings created during the pre-training of the graph.

- The RL Coordinator transmits those jobs to the corresponding agent along with the current machine state received from the MES.

- The agent uses either the neural network or a dispatching rule[7]. If a neural network is used, the similarity score between the job embedding and the machine state is calculated for each job. The job with the highest similarity score is then returned to the RL Coordinator.

[7]Note that, in case the current algorithm type is 'FIFO', 'SPT', or 'random', it does not use the neural network but instead picks the job directly based on the rule to be applied. This is not shown in the figure for simplicity

- The RL Coordinator sends a message towards the MES, which then sends this job to the corresponding machine. This message is stored in an Event-Queue sorted by their timestamp in order to ensure a correct execution order.
- The machine sends an MQTT Message back to the MES with the resulting timestamp. This event is again stored in the Event-Queue to ensure a correct order of Event execution. The machine calculates the resulting timestamp based on the provided global time and the job duration. Note that potential deviations from the expected job duration are calculated at this step.
- When the Event described above is executed, the MES sends an MQTT Message to the RL Coordinator with the information that the machine is idle again and can perform the next job, along with the new state of the machine. This state consists of the features listed in section 4.2.3. In parallel, the tuple for the replay buffer is created as described in section 4.1 and stored in a separate file.
- When the RL Coordinator receives this message, the described loop starts again. Note that, at this point, the RL Coordinator checks for each agent if it is idle. Due to the imposed constraints regarding job dependencies, it is possible that other machines can now perform jobs as those constraints have been resolved.
- Before every 25th episode, the RL Coordinator automatically starts to (re-)train the neural network used by the agents to calculate the similarity scores. The replay buffer file created by the MES is loaded, and training is initialized. After the training is finished, the episode starts. Note that those episodes were also used for evaluation; throughout every 25th episode, the agents did not act ϵ-greedy, but deterministically. Those episodes were also uploaded to Weights & Biases and the Postgres DB for evaluation.

The next sections provide a more detailed explanation of the relevant parts of this execution pipeline.

4.2.1 RL Coordinator

The RL Coordinator handles the registration phase with the MES, incoming messages, deploying the correct number of agents, and sending messages back to the MES. It further provides the agents with the state of all executable jobs during job Selection.

The **RL Coordinator** represents the central coordination component of the reinforcement learning-based job-shop scheduling system. It acts as the interface between the MES, the machine agents, and the reinforcement learning infrastructure. The RL Coordinator maintains the global system state, manages job assignments, and controls the training process of the multi-agent reinforcement learning MARL-model.

A **Graph State Manager** is used to maintain a graph-based representation of job dependencies. This component generates vector embeddings for executable jobs, which serve as inputs for the reinforcement learning agents during decision making.

job Assignment Process: Whenever a machine becomes idle or new orders arrive, the RL Coordinator attempts to assign a job to the corresponding machine agent and rechecks a potential job assignment for all other machines (in case another machine can now be assigned a job due to a resolved constraint). The assignment procedure consists of the following steps:

- Retrieve the current machine state representation.
- Determine the set of executable jobs using the **JobStateManager**.
- Compute job embeddings for all candidate jobs.
- Let the machine agent select one job from the candidate set.

Depending on the configuration, different job assignment strategies can be used:

- Algorithmic selection
- FIFO
- SPT
- Random selection

When the reinforcement learning strategy is used, the agent receives the machine state representation together with the embeddings of all executable jobs. The agent then selects a job using an ϵ-greedy policy, balancing exploration and exploitation.
After a job has been selected, the RL Coordinator creates a job assignment object containing the machine identifier, job information, and expected processing times. This assignment is transmitted to the simulation environment via MQTT.
Each machine in the system is controlled by an individual reinforcement learning agent, resulting in a multi-agent reinforcement learning setup. All agents share a common neural network model that estimates the value of assigning a specific job to a machine. The RL Coordinator manages the training process using a replay buffer that stores past interactions between agents and the environment.

Experiment Tracking: The RL Coordinator integrates experiment tracking using **Weights & Biases** for monitoring training metrics and model behavior. Additionally, experiment configurations and metadata are stored in a PostgreSQL database to ensure reproducibility and structured experiment management.

Episode Management: The simulation is organized into episodes, each representing one production run. At the end of each episode, the RL Coordinator increments the episode counter and resets the global simulation clock, preparing the system for the next training or evaluation cycle.

The **RL Coordinator** serves as the orchestration layer of the reinforcement learning scheduling system. It connects the simulation environment, machine agents, and training pipeline while maintaining the global system state and coordinating the job assignments. This centralized coordination mechanism enables scalable multi-agent reinforcement learning for dynamic job-shop scheduling environments.

4.2.2 Agents

The **agent** represents the reinforcement learning-based decision-making component that controls a single machine within the scheduling system. Each machine in the Shopfloor simulation is associated with one agent instance, resulting in a multi-agent reinforcement learning setup. The purpose of the agent is to select the most suitable job from the set of currently executable jobs.

Agent Architecture

The agent follows a value-based reinforcement learning approach. Instead of learning a direct policy, the agent relies on a neural network that approximates the action-value function

$$Q(s, a)$$

where

- s represents the current machine state
- a represents a candidate job assignment

The neural network receives two inputs:

- a machine state representation (machine feature vector)
- a job embedding representing the candidate job

The network outputs a scalar value representing the estimated quality of assigning the given job to the machine in the current state. Higher Q-values correspond to more promising scheduling decisions.

This formulation allows the agent to evaluate arbitrary machine–job combinations and generalize across different system states.

4.2.3 Features

The features were derived in the replay buffer Generator mentioned in the overall architecture of the Shopfloor Simulator; see figure 4.1. The features are provided as input for the RL-algorithm and form the basis for the state representation of a machine:

- job execution outcome indicator (bool): Whether the last job was completed successfully.
- Long-term job delay rate of the machine (float): Historical share of delayed jobs on the machine.
- Short-term job delay rate of the machine (float): Share of delayed jobs of the last 3 assignments on the machine.
- Relative delay rate compared to other machines (float): Delay rate relative to other machines.
- Long-term delay on the machine (float): Historical mean delay of jobs on the machine.
- Short-term delay on the machine (float): Mean delay of jobs of the last 3 assignments on the machine.
- Relative delay compared to other machines (float): Delay relative to other machines.
- Time since the last job assignment (float): Time since the previous assignment.
- Time since the last machine delay (float): Time since the previous delay.

It was not the focus of this master thesis to explore different types of features set against different configurations. This is a limitation of this work, also mentioned in section 7.6.

Decision Process

The decision process implemented in the `select_action` method follows a standard **ϵ-greedy policy**, where $\epsilon = 0.2$. This strategy balances exploration and exploitation during learning.

Given a set of executable jobs and their corresponding embeddings, the agent proceeds as follows:

- Verify that the machine is currently idle and capable of executing a new job.
- Determine whether the action should be exploratory or greedy.
- Evaluate all candidate job embeddings using the neural network if exploitation is selected.
- Choose the job with the highest predicted Q-value.

Exploration During exploration, the agent selects a job uniformly at random from the set of available jobs. Exploration is triggered with probability ϵ when the episode is not configured to be deterministic. This allows the agent to discover alternative scheduling strategies and gather diverse experience for training.

Exploitation During exploitation, the agent evaluates all candidate jobs using the neural network. For each job embedding a_i, the neural network computes

$$Q(s, a_i)$$

where s represents the machine state. The job with the highest predicted Q-value is selected:

$$a^* = \arg\max_{a_i} Q(s, a_i)$$

This corresponds to choosing the job that the model predicts will lead to the highest expected long-term reward.

Neural Network Evaluation

For each candidate job embedding, the machine state and job embedding are passed through the neural network model. Since the network expects batched inputs, both vectors are temporarily expanded to include a batch dimension before evaluation.

The model outputs a scalar score representing the estimated Q-value. These scores are collected for all candidate jobs, and the job with the maximum score is selected.

Action Representation

Instead of using discrete action identifiers, actions are represented as continuous vectors in the form of job embeddings. This design enables the agent to operate in environments with a dynamic and potentially large action space. By embedding jobs into a fixed-dimensional vector space, the neural network can evaluate job–machine compatibility more effectively.

Deterministic Evaluation Mode

The agent supports a deterministic decision mode used during evaluation episodes. In this mode, exploration is disabled, and the agent always selects the job with the highest predicted Q-value. This allows the performance of the learned scheduling policy to be evaluated without the influence of random exploration.

Agent State Tracking

The agent maintains an internal state indicating whether the corresponding machine is currently idle. Once a job is selected, the agent transitions to a non-idle state until the execution result of the assignment is reported by the simulation environment. Additionally, the identifier of the last selected job is stored for later reference during environment updates.

Summary

The **agent** implements a value-based reinforcement learning policy that selects jobs based on predicted Q-values computed by a neural network. By combining machine state representations with job embeddings, the agent evaluates candidate assignments and selects the job expected to maximize long-term system performance. The use of an ϵ-greedy policy enables a balance between exploration and exploitation during training, while deterministic evaluation allows for consistent policy assessment.

4.2.4 Training Algorithm

The training procedure operates on previously collected transitions stored in the replay buffer and periodically updates the NN which estimates the action-value function. The implementation includes several stabilization techniques commonly used in deep reinforcement learning, i.e. target networks, gradient clipping, reward normalization, and prioritized replay.
The reward is calculated according to Park et al. [13]:

$$\overline{M}(\pi_\theta, \pi_b) = \frac{M(\pi_\theta) - M(\pi_b)}{M(\pi_b)} \tag{4.1}$$

The current makespan is normalized by an idealized lower bound resulting from the total duration of all jobs divided by the number of machines.

Overview of the Training Procedure

The training of the neural network model is performed by samples from a replay buffer. The training process consists of the following stages:

- Lazy initialization of the optimizer and target network
- Loading newly collected experiences
- Sampling batches from the replay buffer
- Computing Q-value predictions for the current state-action pairs
- Approximating the optimal next-state value using an action bank

- Computing temporal-difference targets
- Updating network parameters using gradient descent
- Soft updating the target network
- Updating replay buffer priorities
- Logging and check-pointing the model

This procedure is repeated for a predefined number of training iterations. In the final setup, 20 iterations were used.

Offline Reinforcement Learning Setting

The learning algorithm follows the paradigm of offline reinforcement learning. Instead of learning directly from interactions with the environment in real time, the agent learns from a dataset of previously collected transitions stored in a replay buffer.

Each transition stored in the buffer contains:

- the previous state s
- the selected action a
- the obtained reward r
- the resulting next state s'
- a terminal indicator d

The objective of the algorithm is to approximate the **action-value function**

$$Q(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q(s', a')]$$

which represents the expected cumulative reward when executing action a in state s and following the optimal policy thereafter.

Suitability of Q-learning for the Proposed Approach

Q-learning is particularly well-suited for the considered job-shop scheduling problem due to its compatibility with an offline training paradigm, which is essential in industrial environments where continuous online learning is impractical. In real-world Shopfloor operations, exploratory behavior during learning can lead to suboptimal or infeasible decisions, resulting in delays, increased operational costs, or violations of constraints. By contrast, Q-learning can be trained in a simulated or historical setting, allowing the

agent to learn effective decision policies without interfering with live production processes. Once training is complete, the learned policy can be deployed in a purely exploitative manner, ensuring stable and predictable decision-making.

A further advantage of this approach lies in the ability to explicitly control the training data distribution. Historical datasets can be curated prior to training, enabling the removal of anomalous or non-representative events, such as extreme outlier days caused by rare disruptions. This ensures that the learned policy captures typical system behavior rather than being biased by irregular occurrences. In contrast, online learning approaches continuously incorporate streaming data, making it significantly more difficult to filter such outliers and to maintain consistent learning dynamics over time.

In addition, Q-learning, combined with neural network–based function approximation, enables generalization across high-dimensional state spaces by learning state–action value functions. This is particularly important in complex scheduling environments, where enumerating all possible states is infeasible. The off-policy nature of Q-learning further supports the separation between exploration during training and exploitation during deployment, resulting in a controlled, reproducible, and practically applicable learning framework for industrial decision-making.

Lazy Initialization of Training Components

To avoid repeatedly constructing training objects, the optimizer and target network are initialized lazily during the first training call.

The optimizer used is the Adam optimizer:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L$$

with a learning rate $\alpha = 10^{-4}$.

A separate **target network** is created as a deep copy of the main model. The target network is used to compute stable temporal-difference targets and is updated slowly during training.

Reward Normalization

Before computing the training targets, rewards are normalized by dividing them by the mean absolute reward value in the batch:

$$r_{\text{norm}} = \frac{r}{\max(\epsilon, \mathbb{E}[|r|])}$$

Here, ϵ is a small constant used to avoid division by zero. This normalization reduces scale differences between reward signals within a batch and improves the numerical stability of the training process. It should be distinguished from the makespan-based

reward scaling described above: the reference makespan provides an instance-dependent scale for the reward, whereas the batch-wise normalization stabilizes the magnitude of the rewards during gradient-based training.

Current Q-Value Estimation

The neural network predicts the Q-values for the state-action pairs in the batch:

$$Q_\theta(s, a)$$

These predictions represent the current estimate of the expected cumulative reward (= the Q-value estimation) for each observed assignment decision.

Action Bank Approximation

In classical Q-learning, the target value requires the computation

$$\max_{a'} Q(s', a').$$

For discrete action spaces, this maximum can be computed by evaluating all possible actions in the successor state. However, in this problem, the action space is represented by continuous job embeddings. Therefore, enumerating all possible actions is infeasible.

This representation is necessary because the set of executable jobs is not fixed. It changes depending on the current state of the Shopfloor, the predecessor constraints of the orders, and the machines on which a job can be executed. A fixed action table, as used in classical tabular Q-learning, would therefore not be suitable for the dynamic and flexible JSSP.

During actual Scheduling, the agent does not select arbitrary points from the continuous embedding space. Instead, the Shopfloor environment first determines the set of currently executable jobs. Only the embeddings of these feasible jobs are evaluated by the agent. The selected action is therefore always restricted to a valid scheduling decision.

The replay buffer stores transitions of the form

$$(s, a, r, s', d),$$

where s denotes the current machine state, a denotes the selected job embedding, r denotes the reward, s' denotes the subsequent machine state, and d indicates whether the terminal state was reached. However, the full set of feasible successor actions is not explicitly stored for every sampled transition.

To approximate this maximum, the algorithm constructs an **action bank**. This bank contains candidate actions generated from:

- unique actions observed in the batch [8]
- additional sampled actions drawn from a Gaussian distribution around the batch statistics

The unique actions observed in the batch ensure that the target computation is based on job embeddings that actually occurred during training. The additional sampled actions extend this set locally and reduce the risk that the maximum is computed only over a very small subset of observed batch actions. The action bank, therefore, represents a set of plausible job embeddings in the local embedding space.

The result is a sampled approximation of the successor-action maximum:

$$\max_{a' \in A_{\text{bank}}} Q_{\theta^-}(s', a').$$

Here, A_{bank} denotes the action bank and Q_{θ^-} denotes the target network.

It is important to distinguish between the use of the action bank during training and the actual action selection during Scheduling. The sampled actions from the action bank are only used for approximating the temporal-difference target. They are not executed in the environment. During deployment and evaluation, the agent still evaluates only the currently executable jobs provided by the Shopfloor environment.

Thus, the action bank should be interpreted as a computational approximation that makes offline Q-learning feasible with continuous job embeddings. It does not extend the actual set of feasible scheduling actions.

Target Value Computation

For each subsequent state s', all actions in the action bank are evaluated using the target network. The maximum predicted value is selected:

$$\max_{a' \in A_{bank}} Q_{\theta^-}(s', a')$$

The temporal-difference target is then computed as

$$y = r + \gamma(1 - d) \max_{a'} Q_{\theta^-}(s', a')$$

where

- r is the reward

[8] The batch size was set to 32

- γ is the discount factor
- d indicates terminal transitions

Q-values are clipped to a fixed range to avoid exploding targets.

Loss Function

The training objective consists of two components.

Temporal-Difference Loss The main loss is the Huber loss between the predicted Q-values and the computed targets:

$$L_{TD} = \text{Huber}(Q_\theta(s, a), y)$$

The Huber loss is more robust to outliers than the mean squared error and improves training stability.

Policy Advantage Term An additional policy improvement term has been added:

$$L_{policy} = -\mathbb{E}[(y - Q(s, a))_+ \cdot Q(s, a)]$$

where $(x)_+ = \max(x, 0)$.

The network is encouraged to increase the predicted Q-values for advantageous actions, thus guiding the policy toward better decisions.

The final loss is given by

$$L = L_{TD} + 0.1 L_{policy}$$

Gradient Update and Stabilization

The network parameters are updated via backpropagation. To avoid instability during training, several stabilization techniques are applied:

- Gradient clipping
- Q-value clipping
- A target network

Gradient clipping ensures that excessively large gradients do not destabilize the training process.

Soft Target Network Updates

Instead of copying the main network to the target network periodically, the implementation uses **soft updates**:

$$\theta^- \leftarrow (1 - \tau)\theta^- + \tau\theta$$

with an update rate $\tau = 0.005$.

This technique provides smoother target updates and reduces oscillations during training.

Prioritized Replay Updates

After each training step, the temporal-difference error is computed:

$$\delta = |Q(s, a) - y|$$

These errors are used to update the priorities of the sampled transitions in the replay buffer. Transitions with higher errors will be sampled more frequently in future training steps.

This mechanism is known as **prioritized experience replay** and improves learning efficiency by focusing training on informative samples.

Logging and Model Check-pointing

Training metrics such as loss values, Q-value statistics, and reward averages are logged using **Weights and Biases (wandb)**. This allows monitoring of training progress during experiments.

After each training cycle, the current model parameters are saved and stored as an artifact. This enables reproducibility and later analysis of the training process.

Convergence Detection

Finally, the training procedure evaluates whether convergence has been reached using a makespan-based convergence criterion. If the system performance stabilizes across recent episodes, training can be considered converged.

Graph State Manager

The Graph State Manager provides an interface with the following functionality:

- **Set Machines**: At the beginning, the Graph State Manager is provided with the list of machines. The machines are added to the Graph.

- **Add Order**: adds an order with their corresponding jobs to the Graph.
- **Cancel order**: Cancels an order by marking the corresponding jobs as canceled.
- **job Finished**: Marks a job in the Graph as finished.
- **Get executable jobs By Machine ID**: Returns a list of executable job IDs along with their respective embeddings.

CHAPTER 5

Experimental Setup

This chapter describes the experimental setup used to answer RQ1 and RQ2. First, the generated JSSP instances are defined. Afterwards, the learning setup, the architecture configurations, the RQ-specific experimental designs, and the evaluation protocol are described.

The goal of this chapter is to define the experimental instances and model configurations precisely enough to make the reported results reproducible. In the following, the term instance refers to one generated JSSP instance consisting of a set of orders, their jobs, the available machines, the predecessor relations, the machine eligibility sets, and the corresponding static or dynamic processing-time properties.

5.1 Experimental Pipeline

5.1.1 Terminology

An instance size is denoted as $m \times n$, where m is the number of machines and n is the total number of order in the instance.

Each order consists of a sequence of jobs. Since predecessor constraints are enabled in the experiments, the jobs of an order form a linear precedence chain. The first job of each order is executable at the beginning of the simulation. Every following job becomes executable only after its direct predecessor has been completed.

An experimental configuration is defined by the instance size, the neural network architecture, the static instance-generation parameters, the dynamic-factor parameters where applicable, and the assignment method used during evaluation.

5.1.2 Overall Pipeline

The experimental pipeline consists of the following steps:

- Generate static flexible JSSP instances for the three tested instance sizes.
- Train and evaluate different MLP and CNN architectures on the static flexible instances for RQ1.
- Select the best-performing architecture from RQ1 as the static reference model for RQ2.
- Generate dynamic flexible JSSP instances by introducing either machine delays or job delays.
- Train one dynamic model per dynamic factor type.
- Compare the dynamic model against the static model and the rule-based assignment methods.
- Evaluate the results using makespan and utilization.

RQ1 is used to analyze the effect of the downstream neural-network architecture. RQ2 is used to analyze whether the MARL algorithm can adapt to the introduced dynamic factors.

5.2 Instance Generation Protocol

5.2.1 Instance Sizes

The experiments use three instance sizes. The notation $m \times n$ denotes m machines and n orders in total.

Instance size	Number of machines	Total number of orders
5×15	5	15
10×25	10	25
25×200	25	200

Table 5.1: Instance sizes used in the experiments.

The total number of jobs is randomly distributed across the generated orders. Each order must contain between six and eight jobs.

5.2.2 Precedence Structure

Predecessor constraints are enabled for all experiments. Within each order, jobs form a sequential chain. Let

$J_{o,1}, J_{o,2}, \ldots, J_{o,n_o}$

be the jobs of order o. Then job $J_{o,i+1}$ may only start after job $J_{o,i}$ has finished. Therefore, the precedence constraint within each order is

$s_{o,i+1} \geq c_{o,i}$.

Only the first job of each order is executable at the beginning of an episode. All later jobs become executable only after their predecessor has been completed.

5.2.3 Processing-Time Generation

Clustered Base Processing Times

The base processing time of each job is sampled from a clustered distribution. The cluster centers are

$C = 200, 400, 600, 800.$

For each job, one cluster center $c \in C$ is selected uniformly. Afterwards, a raw processing time $\tilde{p}$ is sampled from a normal distribution around the selected cluster center:

$\tilde{p} \sim \mathcal{N}(c, 30^2)$.

The sampled value is converted to an integer and bounded from below by a minimum processing time of 50:

$p = \max(50, \lfloor \tilde{p} \rfloor)$.

This creates four processing-time groups around 200, 400, 600, and 800 time units. The distribution therefore contains structured variation between shorter and longer jobs, while avoiding a purely uniform spread of processing times.

Long Processing-Time Outliers

In addition to the clustered base processing-time distribution, long-running jobs are introduced by manipulation of existing jobs. For each instance, 20% of the jobs are selected as long-running jobs. Let J be the set of all jobs in one instance and let $J_{\text{out}} \subset J$ be the selected set of long-running jobs.

The remaining jobs are used to compute a reference processing-time average. Let

$J_{\text{ref}} = J \setminus J_{\text{out}}$

be the set of non-outlier jobs.

For each long-running job $j \in J_{\text{out}}$, a factor u_j is sampled uniformly from the interval $[2, 5]$:

$u_j \sim \mathcal{U}(2, 5)$.

The processing time of the long-running job is then set to

$p_j = \lceil u_j \cdot \bar{p}_{\text{ref}} \rceil$.

This mechanism creates jobs whose scheduling position has a stronger effect on the final makespan. The same outlier-generation mechanism is used for RQ1 and RQ2.

5.2.4 machine Eligibility and Flexibility

All final experiments are conducted on flexible JSSP instances. For each job, a subset of eligible machines is sampled. The eligible subset contains 30% of the available machines, rounded up.

Let M be the set of machines and let $m = |M|$. For each job j, the number of eligible machines is

$k = \lceil 0.3m \rceil$.

The machine eligibility set $M_j \subseteq M$ is then sampled uniformly without replacement, with

$|M_j| = k$.

The resulting number of eligible machines per job is shown in Table 5.2.

Instance size	Number of machines	Eligible machines per job
5×15	5	2
10×25	10	3
25×200	25	8

Table 5.2: Number of eligible machines per job induced by the 30% flexibility setting.

5.3 Learning Setup

5.3.1 State Representation and Candidate Scoring

The MARL algorithm uses a shared-policy setup. Each machine is represented by an agent, but all agents share the same neural network parameters. A central coordinator maintains the global Shopfloor state, determines the currently executable jobs, manages the replay buffer, and coordinates training.

At each decision point, an idle machine receives the set of currently executable jobs that can be processed on that machine. For every candidate job, the network receives two inputs: the feature vector of the current machine and the embedding of the candidate job. The machine feature vector is encoded by the neural network. The resulting machine representation is concatenated with the job embedding.

The job with the highest score among the currently executable candidate jobs is selected as the next assignment for the machine.

The final dynamic configurations uses 9 machine features and a job embedding with a size of 4.

5.3.2 MLP Similarity Network

The MLP similarity network receives a concatenated vector of machine features and job embeddings. The layers sizes (depending on the number of layers in a configuration) are:

- 32
- 32, 64, 32
- 32, 64, 96, 64, 32
- 32, 64, 96, 128, 96, 64, 32
- 32, 64, 96, 128, 160, 128, 96, 64, 32

This maximum number depends on the defined number of hidden layers. Each layer consists of a linear layer, followed by a ReLU activation and dropout.

5.3.3 CNN Similarity Network

The layer sizes are designed as with the MLP. Furthermore, each CNN block consists of a one-dimensional convolution, a ReLU activation, a pooling operation, and dropout. The convolution uses padding of size $\lfloor k/2 \rfloor$, where k is the kernel size. The tested kernel sizes are 3 and 5. The tested pooling operations are max-pooling and average-pooling, both with pooling size 2 and stride 1.

After the convolution, the output is averaged over the remaining sequence dimension.

5.3.4 Shared Training Parameters

The shared training parameters used in the experiments are shown in Table 5.3.

RQ1 and RQ2 are each trained for 10,000 episodes. During training, every 25th episode is executed deterministically and used for evaluation. These deterministic evaluation episodes are the episodes reported in the result analysis.

Training is not stopped early based on visual inspection of the learning curves. Instead, the models are trained for the fixed episode budget. The deterministic evaluation curves are inspected after training to assess whether the makespan has stabilized and whether further improvements are visible within the fixed training horizon.

Parameter	Value
machine feature dimension	9
job embedding dimension	4
Epsilon	0.2
Batch size	32
Discount factor γ	0.99
Learning rate	10^{-4}
Optimizer	Adam
Loss function	Huber loss
Epochs	20
Soft target update rate τ	0.005
Reward function	gap

Table 5.3: Shared training parameters.

5.4 Experimental Design for RQ1

5.4.1 Objective

RQ1 analyzes how the architectural hyperparameters and regularization of the neural network used by the MARL agents affect convergence on the flexible JSSP. The experiments for RQ1 are conducted on static flexible JSSP instances. No dynamic delays are included in RQ1. This isolates the effect of the downstream neural-network architecture before dynamic factors are introduced in RQ2.

5.4.2 Training and Test Instances

For each instance size, a total of 500 problem instances were generated. Of these, 450 instances were used for training and 50 separate instances were reserved for testing. At the beginning of each training episode, one instance was selected randomly from the pool of 450 training instances. The 50 test instances were not used during training and were evaluated only after the training process had been completed.

5.4.3 MLP Hyperparameter Grid

For the MLP architecture, the varied hyperparameters are dropout and the number of layers. The tested values are shown in Table 5.4.

Hyperparameter	Tested values
Dropout	$0.1, 0.2$
Number of layers	$1, 2, 3, 4, 5$

Table 5.4: MLP hyperparameter grid for RQ1.

5.4.4 CNN Hyperparameter Grid

For the CNN architecture, dropout, number of layers, kernel size, and pooling type are varied. The tested values are shown in Table 5.5.

Hyperparameter	Tested values
Dropout	$0.1, 0.2$
Number of layers	$1, 2, 3, 4, 5$
Kernel size	$3, 5$
Pooling	max-pooling, average-pooling

Table 5.5: CNN hyperparameter grid for RQ1.

5.4.5 RQ1 Evaluation

RQ1 is evaluated on the three instance sizes 5×15, 10×25, and 25×200. Makespan is used as the main evaluation criterion for comparing the architecture and hyperparameter configurations.

Each experimental configuration is evaluated over 50 runs per combination with different seeds. During training, deterministic evaluation is performed every 25th episode. The deterministic evaluation episodes are used to analyze the learning behavior and to compare the tested architectures.

5.5 Experimental Design for RQ2

5.5.1 Objective

RQ2 analyzes whether the graph-based MARL scheduling algorithm can adapt to dynamic factors. The two dynamic factors investigated are machine delays and job delays. The dynamic factors are tested separately.

The best-performing architecture from RQ1 is used as the static reference model. For each instance size, one dynamic model is trained per dynamic factor type. Therefore, the dynamic model is trained for machine delays as one dynamic-factor type and for job delays as another dynamic-factor type.

The dynamic model is compared against the static model and the rule-based assignment methods FIFO, SPT, and Random. A lower makespan of the dynamic model compared to the static model is interpreted as evidence that the model adapted to the corresponding dynamic factor.

5.5.2 Assignment Methods

The following assignment methods are evaluated in RQ2:

- Dynamic MARL model
- Static MARL model
- FIFO
- SPT
- Random

At each decision point, only jobs that are executable and eligible for the currently idle machine are considered. FIFO selects the executable candidate job with the smallest job index in the generated instance. SPT selects the executable candidate job with the shortest processing time among the currently executable and eligible candidate jobs. Random samples uniformly from the executable candidate jobs of the currently idle machine.

5.5.3 Machine Delay Experiments

Machine delays model delays that are connected to specific machines. For each instance, a subset of affected machines is selected uniformly without replacement. Once selected, the affected machines remain fixed for that instance.

For each assignment to an affected machine, a Bernoulli trial with probability α determines whether the delay is applied. If the delay is applied, the processing time is divided by a constant delay factor f:

$p\_\text{actual} = \frac{p}{f}$.

Since the tested values of f are smaller than 1, this increases the actual processing time. For example, if $f = 0.8$, then

$p\_\text{actual} = 1.25p$.

The varied machine-delay parameters are the number of affected machines, the delay probability α, and the constant delay factor f. The tested parameter values are shown in Table 5.6.

Instance size	Affected machines	α	Constant delay factor f
5×15	$1, 3$	$0.5, 1.0$	$0.8, 0.9$
10×25	$1, 5$	$0.5, 1.0$	$0.8, 0.9$
25×200	$1, 5$	$0.5, 1.0$	$0.8, 0.9$

Table 5.6: machine-delay parameter grid for RQ2.

5.5.4 Job Delay Experiments

Job delays model delays that are connected to specific jobs rather than specific machines. A percentage of jobs is selected as affected over all jobs in the instance. Affected jobs are selected uniformly without replacement.

Let n be the total number of jobs in the instance and let ρ be the affected job percentage. The number of affected jobs is

$n\_\text{delay} = \left\lfloor n \cdot \frac{\rho}{100} \right\rfloor .$

Let d be the configured delay percentage. If a job is affected, its actual processing time is

$p\_\text{actual} = p \cdot \left(1 + \frac{d}{100}\right) .$

For example, a delay percentage of 25 results in

$p\_\text{actual} = 1.25p.$

The varied job-delay parameters are the percentage of affected jobs and the delay percentage. The tested parameter values are shown in Table 5.7.

Instance size	Affected jobs (%)	Delay percentage (%)
5×15	$25, 50, 75$	$25, 50, 75$
10×25	$25, 50, 75$	$25, 50, 75$
25×200	$25, 50, 75$	$25, 50, 75$

Table 5.7: job-delay parameter grid for RQ2.

5.5.5 Combination of Dynamic Factors

The simultaneous combination of machine delays and job delays is only evaluated if both dynamic factors can be learned reliably in isolation. If one of the two dynamic factors cannot be learned reliably on its own, then a combined experiment would not allow a clear interpretation of the results.

5.6 Evaluation

5.6.1 Makespan

The primary evaluation metric is makespan. The makespan measures the time required to complete all jobs in an instance. Lower makespan values indicate better scheduling performance.

Let c_j be the completion time of job j and let s_j be the start time of job j. The makespan is computed as

$C\_\max = \max_{j \in J} c_j - \min_{j \in J} s_j.$

5.6.2 Utilization

Utilization is used as an additional evaluation metric. It is defined as the average utilization over all machines. Utilization is interpreted together with makespan.

5.6.3 Evaluation Runs

Each experimental configuration is evaluated over 50 runs. This applies to both RQ1 and RQ2. The reported results are based on the deterministic evaluation episodes. These episodes are executed every 25th training episode.

For RQ2, the result tables report median values. Boxplots are used to analyze the spread of the distributions, the overlap between assignment methods, and shifts in the median. This is important because mean values alone may hide whether the performance difference is stable or caused by a small number of extreme runs.

5.6.4 Comparison Logic

For RQ1, the architecture comparison is based on the deterministic evaluation makespan.

For RQ2, the dynamic model is compared against the static model and the rule-based assignment methods. The comparison follows three principles.

- If the dynamic model achieves a lower makespan than the static model, this indicates that the dynamic model adapted to the tested dynamic factor.
- If the static model achieves a lower makespan than the rule-based assignment methods, this indicates that the static learned policy transfers useful scheduling behavior to the dynamic setting.
- If a rule-based assignment method achieves the best makespan, this indicates that the learned policy did not exploit the tested dynamic factor better than the corresponding simple rule.

The interpretation does not rely only on mean makespan values. The spread of the results, the overlap between distributions, and the utilization values are also considered.

CHAPTER 6

Results and Interpretations

6.1 RQ1

The goal of RQ1 was to analyze whether the architecture of the neural network used by the agents has an effect on convergence. Furthermore, it was analyzed whether this behavior changes for different instance sizes. The tested architectures were MLPs and CNNs. For both architectures, different combinations of dropout and the number of layers were tested. For CNNs, additionally, the kernel size and the pooling type were adjusted.

Different hyperparameters and architectures have been trained and compared. A total of 50 different orders were randomly created; 40 of them were used during training, and 10 were used for testing.

RQ1 was tested on the static and flexible JSSP. This means that no dynamic factors, such as machine delays or job delays, were included. The reason for this is that RQ1 should first test whether the algorithm can learn the flexible JSSP without any additional uncertainty. The results from RQ1 were then used to decide which architecture should be used for RQ2.

6.1.1 RQ1.1: MLP Architectures

MLP architectures showed the best results in the tested setup. While none of the configurations with dropout=0.2 and more than 96 neurons in a hidden layer showed clear signs of improvement in terms of makespan optimization, using a dropout of 0.1 and a maximum of 64 neurons in a hidden layer led to the best makespan and showed the clearest sign of convergence.

A consistent pattern was that adding more layers did not improve the result. This is important because one could expect that a deeper network is able to model the decision

process better. However, this was not the case in the experiments conducted. The best results were achieved by a small network. This indicates that the problem, after the graph embedding was created, does not require a very deep, fully connected network.

One possible explanation is that the graph embedding already encodes a large part of the structural information of the JSSP. The MLP then only needs to compare the current machine state with the available jobs. Therefore, a small MLP may already be sufficient. Adding more layers may only make the training less stable without adding useful information.

The dropout also had a clear effect. A dropout of 0.2 was too strong in the tested setup. It did not lead to better generalization but rather prevented the model from learning useful scheduling behavior. A dropout of 0.1, on the other hand, worked better. This indicates that some regularization is useful, but too much regularization hurts the learning process.

The author argues that this is also related to the size of the training setup. The algorithm was trained on a fixed number of generated orders. If the network is too large or regularized too strongly, the model may not be able to learn the relevant patterns in a stable way. Therefore, for the given setup, the best MLP architecture was not the largest one, but the one with 2 layers and dropout=0.1.

6.1.2 RQ1.2: CNN Architectures

None of the CNN architectures converged. This pattern was consistent across all tested JSSP sizes. Only configurations with dropout=0.1, kernel_size=3, a maximum layers size of 96 and average pooling showed slight, but non-significant improvements in terms of stabilization of the makespan, which indicated some level of learning. However, even those configurations performed worse than the best MLP architectures. Due to the poor results, especially compared to MLP architectures, no further examinations of CNN architectures were conducted for RQ2.

This result is surprising in the sense that other works have been able to use CNN architectures in their static JSSP algorithms, as shown in chapter 3. However, the input representation used in this thesis is different. CNNs are usually useful if neighboring input values have a meaningful relation. This is the case for images, where pixels close to each other are also related to one another. In this work, the input is based on graph embeddings and machine/job features. The order of these features does not necessarily have such spatial meaning.

Therefore, it is possible that the CNN did not fit the representation used. The Convolutional layers may search for local patterns that are not actually meaningful in the given input. This may also explain why max-pooling performed poorly. Max-pooling removes information by keeping only the maximum activation in a local window. If the local window itself does not have a clear meaning, this may remove useful information instead of improving the representation.

Average pooling performed slightly better than max-pooling, probably because it is less aggressive. However, this did not change the general result that CNN architectures were not suitable for the chosen representation.

A more detailed interpretation of those results is difficult without tuning other factors, such as the graph embedding or the feature set. This is a limitation of this work and is further examined in section 7.6.

6.1.3 RQ1.3: Comparison between MLPs and CNNs

The comparison between MLPs and CNNs shows a clear difference. MLP architectures were able to learn the static flexible JSSP, while CNN architectures mostly did not converge. The best MLP architecture used 2 layers and dropout=0.1. CNN architectures only showed small improvements under very specific hyperparameter combinations and were therefore not used further.

The author argues that this difference is mainly caused by the representation. The graph embedding already creates a representation of the JSSP structure. After this step, the neural network of the agents does not need to learn the full structure of the JSSP again. It rather needs to learn a scoring function between the machine and the available jobs. For this task, an MLP seems to be sufficient.

A CNN, on the other hand, adds an assumption about local relationships in the input. This assumption may not be valid for the used feature representation. Therefore, the CNN may not be able to use its main strength.

Furthermore, the makespan dropped within the first 2000 episodes and stabilized afterwards. However, the improvement was only around 15% lower than the first makespan on a model without any previous knowledge.

After further examination, this behavior is explained by the problem definition itself:

- By design, the agent is only represented with jobs that can be executed on that machine. Therefore, even if the algorithm picks a bad option, it still picks a feasible option.
- To improve the makespan, there must be room for improvement. This may sound trivial, but it is important for the static JSSP. For example, in a 5x15 instance, each machine executes only a small number of jobs on average. While the optimal solution may be found quickly, by design, the makespan cannot drop by much.

When trying to mitigate these two factors, it is necessary to re-design the chosen jobs and their duration, and not pick them purely at random. Really bad options for decisions can be created with heavy outliers. Such outliers were identified in this thesis after the first iteration of experiments. Those outliers were detected by the best hyperparameters after a few hundred training iterations.

This shows a very important aspect of the static JSSP: If and how makespan can be reduced is heavily impacted by the problem definition itself. The author of this work argues that this aspect is not sufficiently represented in the current literature.

Using this aspect, it becomes clear that comparing the literature in this area is very challenging. Only some of the works expose their hyperparameters, specify the exact durations of their used jobs, or highlight this issue. The author of this work argues that the static JSSP should be created based on a real-life scenario. How an instance can be solved and which features are needed is problem-specific and very challenging to generalize. This problem is further discussed in section 7.4 and section 7.6.4.

In essence, for the algorithm to learn, there needs to be room for a few big mistakes. What these options for mistakes are is not defined across the literature, leading to a lack of reasonable comparison.

In this work, around 20% of the jobs had durations 2 to 5 times higher than the average of the remaining 80% of the jobs. This was also done for RQ2. Even using solely this option for mistakes must be designed. If the spread between the jobs is too high, the makespan is mostly defined by those extreme outliers. If the spread is too low, there is too little room for mistakes. This problem was expected to be mitigated in RQ2 due to its stochastic behavior.

Furthermore, this showcases the difference between optimality and relevance in the context of the JSSP. Depending on the job durations defined, a schedule can be optimal and still not significantly better than other schedules. By defining heavy outliers with job dependencies, there is room for clearly better schedules, even if they are not necessarily optimal.

Overall, RQ1 can be answered as follows: For the tested graph-based MARL approach, MLP architectures were more suitable than CNN architectures. The best results were achieved with 2 layers and a dropout of 0.1. Increasing the depth of the network or using a stronger dropout did not improve the results. CNN architectures did not fit the chosen representation and were therefore not used for RQ2.

6.2 RQ2

For RQ2, the best model from RQ1 was used as the starting point for further training. In this section, this model is referred to as the *static algorithm*. The same model was then trained further in dynamic environments. This further trained model is referred to as the *dynamic algorithm*.

For each experimental configuration, 50 runs with different random seeds were conducted. The random seeds influenced the stochastic behavior during runtime; for example, whether a delay occurred in a specific situation.

In addition to the median values reported in the tables of this section, the appendix contains box plots for each configuration. These plots are important because stochastic

delays can lead to substantial variation between runs. Therefore, the interpretation does not only consider the mean performance but also the spread of the distributions, the overlap between algorithms, and whether the dynamic algorithm shows a consistent shift in median performance.

The results are compared against three heuristic baselines: FIFO, SPT, and Random. In the plots, each figure consists of five groups, ordered from left to right as follows:

- dynamic algorithm
- static algorithm
- FIFO
- SPT
- Random

Each group contains two or three boxplots, depending on the experiment type. For the machine delay experiments, each configuration (affected machines, alpha, constant factor) has two levels. Therefore, two boxplots are shown per group. For the job delay experiments, each configuration (affected jobs pct, delays job pct) has three levels. Therefore, three boxplots are shown per group. The boxplots can be found in the appendix.

The first set of experiments investigated delays caused by machines. The varied factors were the number of affected machines, the probability α that a processed job would be delayed, and the constant delay factor. The median results for all factorial combinations are shown in Tables 6.1, 6.2, and 6.3.

6.2.1 Machine Delays

Across the machine-delay experiments, the dynamic policy consistently improved over the static policy. In all 24 machine-delay scenario medians, the dynamic policy achieved a lower reactive makespan and a higher utilization than the static policy. The median reactive makespan improvement of the dynamic policy over the static policy was 1.18%, with a mean improvement of 1.72%.

However, the dynamic policy did not dominate all heuristic baselines in every scenario. It achieved the best reactive makespan in 21 of 24 machine-delay scenarios and the highest utilization in 18 of 24 scenarios. Therefore, the machine-delay results should be interpreted as a strong and consistent improvement over the static policy, but not as universal dominance over every baseline in every configuration.

The boxplots in the appendix support this conclusion. For machine delays, the dynamic algorithm is not only better in terms of mean performance, but its makespan distributions are generally shifted below those of the static algorithm and the heuristic baselines. The

Delay factor	Affected machines	α	Algorithm	Reactive makespan	Utilization (%)
0.9	1	0.5	Dynamic	18732.50	85.29
			Static	18808.69	84.36
			FIFO	18927.50	84.78
			SPT	19337.50	82.33
			Random	18908.50	85.97
		1.0	Dynamic	18834.52	85.42
			Static	19201.67	83.92
			FIFO	19211.50	84.71
			SPT	19305.50	83.26
			Random	18946.50	84.06
	3	0.5	Dynamic	18974.30	85.95
			Static	19197.40	84.97
			FIFO	19280.50	84.65
			SPT	19615.50	83.98
			Random	19102.00	86.00
		1.0	Dynamic	19636.26	85.73
			Static	20009.26	83.89
			FIFO	19909.50	84.89
			SPT	20073.00	83.84
			Random	20055.50	84.26
0.8	1	0.5	Dynamic	18920.14	85.34
			Static	19272.11	84.45
			FIFO	18876.00	85.99
			SPT	19614.50	83.17
			Random	18988.00	84.91
		1.0	Dynamic	18960.69	87.04
			Static	19581.97	84.08
			FIFO	19551.50	84.50
			SPT	19853.00	83.22
			Random	19383.50	85.03
	3	0.5	Dynamic	19522.64	87.10
			Static	19816.78	85.33
			FIFO	19959.00	84.28
			SPT	20478.50	82.43
			Random	19857.50	85.74
		1.0	Dynamic	20253.97	89.37
			Static	21018.73	85.26
			FIFO	21042.50	84.90
			SPT	21627.50	84.00
			Random	21250.00	85.33

Table 6.1: machine-delay reactive performance for size 5×15. Reported values are medians over 50 evaluation instances.

Delay factor	Affected machines	α	Algorithm	Reactive makespan	Utilization (%)
0.9	1	0.5	Dynamic	16733.05	79.61
			Static	16760.43	79.10
			FIFO	16484.00	80.64
			SPT	17077.00	78.09
			Random	16272.00	79.83
		1.0	Dynamic	16596.17	80.73
			Static	16666.36	80.14
			FIFO	16683.50	80.59
			SPT	17202.00	77.11
			Random	16743.50	80.46
	5	0.5	Dynamic	16891.48	81.04
			Static	17019.57	80.05
			FIFO	16940.00	80.46
			SPT	17431.00	78.51
			Random	16932.00	81.14
		1.0	Dynamic	17093.26	82.42
			Static	17648.35	79.75
			FIFO	17594.50	79.95
			SPT	18488.50	77.16
			Random	17302.00	80.40
0.8	1	0.5	Dynamic	16572.70	80.68
			Static	16710.14	80.05
			FIFO	16771.00	79.80
			SPT	17168.50	78.35
			Random	16368.00	80.67
		1.0	Dynamic	16749.38	80.71
			Static	17061.18	79.40
			FIFO	17089.50	78.54
			SPT	17515.50	76.55
			Random	16832.00	79.68
	5	0.5	Dynamic	17274.91	81.33
			Static	17808.86	79.59
			FIFO	17882.00	79.41
			SPT	17680.00	78.63
			Random	17343.50	80.07
		1.0	Dynamic	17323.10	85.02
			Static	18221.48	80.74
			FIFO	18411.00	80.49
			SPT	18848.50	76.68
			Random	18411.00	80.07

Table 6.2: machine-delay reactive performance for size 10×25. Reported values are medians over 50 evaluation instances.

Delay factor	Affected machines	α	Algorithm	Reactive makespan	Utilization (%)
0.9	1	0.5	Dynamic	44592.98	94.63
			Static	44635.69	94.42
			FIFO	45053.50	93.71
			SPT	47111.00	89.65
			Random	45425.50	92.65
		1.0	Dynamic	44449.11	95.04
			Static	44634.17	94.65
			FIFO	45217.50	93.35
			SPT	46981.50	89.77
			Random	45665.00	92.89
	5	0.5	Dynamic	45478.46	93.51
			Static	45746.03	92.99
			FIFO	45571.50	93.53
			SPT	47475.50	90.14
			Random	45965.50	92.36
		1.0	Dynamic	45009.30	95.48
			Static	45293.17	94.61
			FIFO	45640.00	93.78
			SPT	47556.00	90.43
			Random	45900.50	93.09
0.8	1	0.5	Dynamic	44609.10	94.93
			Static	44812.81	94.46
			FIFO	45324.50	93.48
			SPT	46953.00	90.09
			Random	45527.00	92.96
		1.0	Dynamic	44756.68	94.89
			Static	44948.64	94.31
			FIFO	45388.00	93.53
			SPT	47058.50	90.11
			Random	45682.00	92.55
	5	0.5	Dynamic	44824.50	95.87
			Static	45420.58	94.35
			FIFO	45917.00	93.54
			SPT	48103.50	89.87
			Random	46352.50	92.47
		1.0	Dynamic	45200.60	97.17
			Static	46320.78	94.70
			FIFO	46855.50	93.71
			SPT	48584.50	89.65
			Random	47292.00	92.43

Table 6.3: machine-delay reactive performance for size 25×200. Reported values are medians over 50 evaluation instances.

same pattern appears for utilization, where the dynamic algorithm usually has a higher median utilization. Although the variance increases over the instance size, the dynamic algorithm remains clearly separated from the baselines in many configurations. This suggests that the improvement is not only caused by a few favorable random seeds, but also reflects a more robust scheduling behavior.

For example, in the 5×15 setting with delay factor 0.8, three affected machines, and $\alpha = 1.0$, the dynamic policy achieved a reactive makespan of 20253.97, compared to 21018.73 for the static policy. The utilization was also higher, with 89.37% for the dynamic policy and 85.26% for the static policy.

A similar pattern appears in the 25×200 setting with delay factor 0.8, five affected machines, and $\alpha = 1.0$. The dynamic policy reached a reactive makespan of 45200.60, while the static policy reached 46320.78. The utilization increased from 94.70% for the static policy to 97.17% for the dynamic policy. These results make sense for the dynamic and flexible JSSP. If specific machines are affected by delays, they become less reliable resources. Since jobs can be processed on alternative machines, the dynamic algorithm can learn to avoid delayed machines where possible. This can reduce waiting times and prevent the delayed machines from becoming strong bottlenecks.

The utilization values support this interpretation. In most configurations, the dynamic algorithm has both the lowest makespan and the highest utilization. This suggests that the dynamic policy keeps the machines active more consistently and reduces idle periods caused by delays. The effect is especially visible for the largest instance size.

The dynamic algorithm also remained robust when the disturbance level increased. When more machines were affected or when the delay probability increased, the reactive makespan generally increased. Utilization, however, is less straightforward: stronger disturbances can either reduce utilization through idle waiting or increase occupied time because delayed jobs keep machines busy longer. Therefore, utilization is interpreted together with makespan rather than as a monotonic severity indicator.

The results also show that the advantage of the dynamic algorithm is not limited to one instance size. It performed best for the small, medium, and large instances. However, the reason for the improvement becomes more important in larger instances. In small instances such as 5x15, the number of routing options is still limited. In larger instances such as 25x200, there are more jobs, more dependencies, and more possible machine assignments.

Compared to the heuristic baselines, the dynamic policy achieved the best result in most, but not all, machine-delay configurations. The results show that the graph-based MARL approach can outperform these rules when the environment contains machine-based stochastic disturbances.

In general, the dynamic algorithm can reduce the makespan if it learns to react to the current state of the Shopfloor instead of following a fixed scheduling policy. This is especially relevant in a flexible JSSP, because jobs can often be assigned to different

machines. If a machine or job causes delays, the scheduling policy can adapt by choosing alternative assignments or by changing the order in which jobs are processed. This can reduce waiting times, avoid bottlenecks, and improve the use of available machine capacity.

Overall, the results show that the dynamic algorithm clearly benefits from additional training in the machine delay setting. In this case, it consistently achieved lower makespans and higher utilization than the static algorithm. Compared to the heuristic baselines, it achieved the best result in most, but not all, configurations. For job delays, the results are less clear. The dynamic algorithm was still competitive and often better than the heuristic baselines, but the improvement over the static algorithm was smaller, although it was consistent across the scenario medians. However, the dynamic policy did not dominate the heuristic baselines in all configurations.

A general relationship between utilization and makespan can be observed in many experiments. Higher utilization often corresponds to lower makespans because machines spend less time idle and the workload is distributed more effectively.

6.2.2 Job Delays

The second set of experiments investigated delays caused by jobs instead of machines. In this setting, a certain share of jobs was affected by job-based delays. If a job was affected, its processing time was multiplied by the corresponding job-delay percentage. The corresponding plots are shown in the appendix. The median results are reported in Tables 6.4, 6.5, and 6.6.

Compared to machine delays, the results for job delays are weaker, but they are not absent. Across all 27 job-delay scenario medians, the dynamic policy achieved a lower reactive makespan and a higher utilization than the static policy. However, the magnitude of this improvement was small. The median reactive makespan improvement over the static policy was only 0.22%, with a mean improvement of 0.28%.

The dynamic policy therefore shows a weak but consistent improvement over the static policy. Nevertheless, it did not dominate the heuristic baselines. It achieved the best reactive makespan in 16 of 27 job-delay scenarios and the highest utilization in 21 of 27 scenarios. This means that job delays appear to be more difficult to exploit than machine delays, even though the dynamic policy still improves slightly over the static policy.

For example, in the 5×15 setting with 25% affected jobs and a job-delay percentage of 25%, the dynamic policy achieved a reactive makespan of 19973.91, while the static policy reached 20007.30. This is an improvement, but the difference is small. In the same configuration, Random achieved a slightly lower makespan of 19970.00, showing that the dynamic policy did not dominate all baselines.

A similar pattern appears in the 10×25 setting with 25% affected jobs and a job-delay percentage of 25%. The dynamic policy slightly improved over the static policy, with a

Affected jobs	job-delay percentage	Algorithm	Reactive makespan	Utilization (%)
25%	25%	Dynamic	19973.91	85.19
		Static	20007.30	85.17
		FIFO	20250.50	85.05
		SPT	20217.50	84.14
		Random	19970.00	84.82
50%		Dynamic	21494.78	84.07
		Static	21532.38	83.91
		FIFO	21462.00	84.32
		SPT	21658.00	83.21
		Random	21010.00	84.85
75%		Dynamic	22394.57	83.98
		Static	22446.79	83.84
		FIFO	22371.00	84.23
		SPT	22835.50	82.77
		Random	22334.00	83.63
25%	50%	Dynamic	21225.36	83.65
		Static	21272.86	83.52
		FIFO	21190.50	83.75
		SPT	21292.50	82.23
		Random	21340.50	83.84
50%		Dynamic	23078.60	85.40
		Static	23204.15	85.24
		FIFO	23341.00	85.27
		SPT	23884.00	83.48
		Random	23578.50	85.15
75%		Dynamic	25533.93	84.61
		Static	25661.45	84.36
		FIFO	25622.00	84.39
		SPT	26377.50	82.85
		Random	25748.50	84.02
25%	75%	Dynamic	22096.36	84.98
		Static	22124.83	84.85
		FIFO	22162.50	83.89
		SPT	23037.00	81.55
		Random	22301.00	84.21
50%		Dynamic	25774.15	84.64
		Static	25857.85	84.34
		FIFO	26236.50	83.48
		SPT	26401.00	82.55
		Random	26018.50	84.02
75%		Dynamic	28926.47	85.10
		Static	29093.32	84.62
		FIFO	29174.50	84.45
		SPT	29745.50	83.16
		Random	28898.00	84.48

Table 6.4: job-delay reactive performance for size 5×15. Reported values are medians over 50 evaluation instances.

Affected jobs	job-delay percentage	Algorithm	Reactive makespan	Utilization (%)
25%	25%	Dynamic	17764.82	79.32
		Static	17775.12	79.23
		FIFO	17650.50	79.40
		SPT	17947.50	79.10
		Random	17804.00	79.14
50%		Dynamic	18535.40	81.20
		Static	18546.32	81.06
		FIFO	18738.50	80.69
		SPT	18842.00	78.49
		Random	18419.00	80.56
75%		Dynamic	19457.73	80.93
		Static	19502.11	80.71
		FIFO	19734.00	79.99
		SPT	20405.00	77.40
		Random	19545.00	78.91
25%	50%	Dynamic	18646.96	80.67
		Static	18681.85	80.59
		FIFO	18647.00	79.71
		SPT	19114.00	77.83
		Random	18603.00	80.00
50%		Dynamic	20811.63	80.03
		Static	20894.97	79.71
		FIFO	20851.00	79.99
		SPT	21583.50	76.47
		Random	20749.50	79.30
75%		Dynamic	22309.83	81.69
		Static	22377.06	81.24
		FIFO	22644.00	80.25
		SPT	23486.50	78.14
		Random	23036.50	78.30
25%	75%	Dynamic	19739.63	79.37
		Static	19787.08	79.21
		FIFO	19706.00	78.40
		SPT	20171.00	77.30
		Random	19844.00	80.04
50%		Dynamic	22470.22	81.40
		Static	22505.50	81.08
		FIFO	22723.00	79.52
		SPT	23664.00	76.39
		Random	23244.50	78.85
75%		Dynamic	25491.54	81.67
		Static	25557.26	81.32
		FIFO	25859.00	80.17
		SPT	26635.00	76.97
		Random	25617.50	80.81

Table 6.5: job-delay reactive performance for size 10×25. Reported values are medians over 50 evaluation instances.

Affected jobs	job-delay percentage	Algorithm	Reactive makespan	Utilization (%)
25%	25%	Dynamic	48006.73	93.05
		Static	48052.43	92.91
		FIFO	47931.00	93.26
		SPT	49843.50	90.00
		Random	48155.00	93.03
50%		Dynamic	50137.85	94.20
		Static	50201.17	94.15
		FIFO	50586.00	93.37
		SPT	52688.00	89.58
		Random	51064.00	92.81
75%		Dynamic	53071.63	93.93
		Static	53223.29	93.73
		FIFO	53475.50	93.49
		SPT	55742.50	89.79
		Random	54297.50	92.16
25%	50%	Dynamic	50724.27	93.48
		Static	50780.26	93.41
		FIFO	51053.50	92.80
		SPT	52934.50	89.58
		Random	51288.50	92.34
50%		Dynamic	55785.08	94.48
		Static	56006.87	94.23
		FIFO	56599.00	93.38
		SPT	58642.00	89.83
		Random	56981.50	92.00
75%		Dynamic	61883.48	93.39
		Static	62078.40	92.90
		FIFO	61927.00	93.14
		SPT	64584.50	89.64
		Random	62679.00	92.38
25%	75%	Dynamic	53374.03	93.76
		Static	53430.55	93.59
		FIFO	53895.50	92.87
		SPT	55899.00	89.31
		Random	54023.00	92.13
50%		Dynamic	61670.83	93.73
		Static	62006.07	93.38
		FIFO	62067.50	93.23
		SPT	64636.50	89.61
		Random	62819.00	92.78
75%		Dynamic	69912.20	93.84
		Static	70376.95	93.35
		FIFO	70584.00	93.07
		SPT	73224.50	89.60
		Random	71202.50	91.90

Table 6.6: job-delay reactive performance for size 25×200. Reported values are medians over 50 evaluation instances.

reactive makespan of 17764.82 compared to 17775.12. However, FIFO achieved a lower makespan of 17650.50.

For the 25×200 setting with 75% affected jobs and a job-delay percentage of 75%, the dynamic policy reached a reactive makespan of 69912.20, compared to 70376.95 for the static policy. This again indicates a consistent improvement over the static policy, but the relative improvement remains small.

A possible explanation is that job delays are harder to learn than machine delays. machine delays are connected to fixed resources. If a machine repeatedly causes delays, this creates a stable pattern in the environment. The agent can learn that this machine should be avoided if alternatives are available. job delays are different. The delayed entity moves through the Shopfloor together with the job. Therefore, the effect of the delay changes depending on the current machine assignment, predecessor constraints, and the position of the job in the schedule.

Increasing the percentage of affected and delayed jobs generally made the problem harder. Across the instance sizes, stronger disturbances usually led to higher makespans and lower utilization. For example, in the 25×200 setting, the dynamic policy reached a reactive makespan of 48006.73 with 25% affected jobs and a job-delay percentage of 25%. With 75% affected jobs and a of 75%, the reactive makespan increased to 69912.20. This shows that stronger job-delay settings increased the makespan, even though utilization did not decrease monotonically.

Overall, the job-delay experiments show that the dynamic policy consistently but only slightly outperformed the static policy in the scenario medians. However, this advantage was small, and heuristic baselines achieved the best makespan in several configurations.

This weaker relationship is also visible in the utilization boxplots in the appendix: for job delays, the utilization distributions of the dynamic and static algorithms often overlap, while the corresponding makespan rankings still change between configurations.

6.2.3 RQ2.1: Difference between Machine Delays and Job Delays

RQ2 asked how many of the introduced dynamic factors can be learned by graph-based MARL scheduling algorithms. More specifically, RQ2.1 asked what the difference between learning machine delays and job delays is.

The results show a clear difference between the two dynamic factors. Machine delays produced a stronger learnability signal than job delays. In the machine-delay experiments, the dynamic policy improved over the static policy in all scenario medians and achieved the best makespan in most configurations. The improvement was also substantially larger than for job delays.

For job delays, the dynamic policy also improved over the static policy in all scenario medians, but the improvement was much smaller. In addition, heuristic baselines achieved the best makespan in several job-delay configurations. Therefore, job delays cannot be

described as completely unlearned. Rather, they should be interpreted as weakly and less robustly learnable in the tested setup.

The difference between both factors is therefore not a binary distinction between “learnable” and “not learnable”. Instead, the results suggest that learnability depends on where the disturbance is located and how actionable the disturbance is for the scheduling policy. Machine delays are tied to fixed resources, so avoiding or deprioritizing affected machines can be a useful strategy. Job delays move through the Shopfloor together with the affected job, which makes their effect more dependent on predecessor constraints, successor jobs, machine eligibility, and the current schedule state.

In summary, machine delays can be learned with the current approach, while job delays cannot be considered learned in the same stable way. This does not mean that the algorithm cannot handle job delays at all. However, the results are not clear enough to argue that the dynamic algorithm learned this factor reliably.

6.2.4 RQ2.2: Why Both Dynamic Factors Were Not Combined

RQ2.2 originally asked how the algorithm behaves if both dynamic factors are learned simultaneously. However, this question only makes sense if both delay types can be learned individually. If one delay type cannot be learned clearly, then combining both factors would make the interpretation of the results difficult.

The results show that machine delays can be learned by the dynamic algorithm. Across the tested instance sizes, the dynamic algorithm achieved a lower makespan and a higher utilization than the static algorithm in all machine-delay scenario medians. Compared to the heuristic baselines, it achieved the best result in most, but not all, machine-delay configurations.

For job delays, this was not the case to the same extent. The dynamic algorithm stayed competitive, but it only slightly outperformed the static algorithm and did not clearly dominate the heuristic baselines. In some configurations, the difference between the dynamic and static algorithm was small. In other configurations, the static algorithm or even heuristic rules achieved better results in several configurations, as shown in Tables 6.4, 6.5, and 6.6.

The combined experiment was not evaluated because the job-delay results were not strong enough to allow a clear attribution of effects in a combined setting. Although the dynamic policy improved slightly over the static policy for job delays, the improvement was small and heuristic baselines still achieved the best makespan in several configurations. If both dynamic factors were combined, it would therefore be unclear whether good results were caused by learning both factors or mainly by exploiting the more stable machine-delay patterns.

CHAPTER 7

Discussion

The results of this thesis show that the performance of Graph-based MARL for the dynamic and flexible JSSP depends on two closely related aspects: the representation of the scheduling state and the type of uncertainty that has to be learned. For RQ1, the experiments indicate that a comparatively small MLP is sufficient for the tested Graph-based representation, while the tested CNN architectures were not suitable under the chosen input structure. For RQ2, the results show a clear difference between machine-based and job-based disturbances. For RQ2, the results show a clear difference between machine-based and job-based disturbances. Machine delays produced a strong and consistent learnability signal. Job delays produced a weaker but still consistent improvement over the static policy. However, the improvement for job delays was small and did not lead to dominance over the heuristic baselines in all configurations.

This distinction is central for interpreting the contribution of the thesis. Dynamic factors in the JSSP are not equally learnable. A dynamic factor that is connected to a fixed resource, such as a machine delay, creates more stable patterns than a dynamic factor that moves through the Shopfloor together with a job. Therefore, the results should not be interpreted as showing that Graph-based MARL can or cannot learn "the dynamic JSSP" in general. Rather, the results suggest that learnability depends on how the dynamic factor is represented, where the disturbance is located in the scheduling system, and whether the generated instances provide enough room for improved scheduling decisions.

7.1 Summary of the Main Findings

The first main finding of this thesis is that larger neural networks were not automatically better for the tested Graph-based MARL architecture. As shown in section 6.1.1, the best results for the MLP architectures were achieved with a rather small network using two layers and a dropout of 0.1. Adding more layers did not improve the results, and stronger regularization did not lead to better generalization. This indicates that, after

the Graph embedding was created, the downstream neural network did not require much additional depth to model the decision problem.

The second main finding is that the tested CNN architectures did not converge well, as shown in section 6.1.2. This should not be interpreted as a general statement that CNNs are unsuitable for the JSSP. Rather, it indicates that CNNs were not suitable for the representation used in this thesis. The input representation did not provide a meaningful spatial structure that could be exploited by Convolutional filters.

The third main finding is that machine delays were learned more reliably than job delays. As shown in section 6.2.1, the dynamic algorithm was able to improve the makespan and utilization in the machine-delay experiments. In contrast, the job-delay experiments showed only a weak learning signal. The dynamic policy improved over the static policy, but the improvement was small and heuristic baselines remained competitive.

The combination of both dynamic factors was not evaluated further. As discussed in section 6.2.4, RQ2.2 can only be answered in a meaningful way if both dynamic factors have been shown to be learnable individually. Since the job-delay signal was weak and less robust than the machine-delay signal, combining job delays and machine delays would not have produced clearly attributable evidence.

7.1.1 Correlation between Makespan and Utilization

Makespan and utilization were strongly related in many of the experiments, especially for machine delays. In those cases, a lower makespan usually implied a higher utilization because the machines spent less time in an idle state. Therefore, in several parts of the results, both metrics led to similar conclusions.

7.2 Analysis of RQ1: Architecture and Representation

The goal of RQ1 was to analyze how the architectural hyperparameters and regularization of Graph-based MARL agents affect convergence on the flexible JSSP. The results indicate that the architecture of the neural network used after the Graph embedding has a direct impact on convergence. However, the results also show that more complex architectures do not necessarily lead to better scheduling behavior.

7.2.1 MLP Architectures

The MLP architectures showed the best results in the tested setup. As discussed in section 6.1.1, the best configuration was a relatively small MLP with two layers and a dropout of 0.1. This is an important result because one could expect that a deeper network would be able to model the decision-making process better. However, this was not the case in the experiments conducted.

One possible explanation is that the Graph embedding already captures a large part of the structural information of the JSSP. The downstream MLP therefore does not need to

learn the full structure of the Shopfloor again. Instead, it mainly needs to evaluate the compatibility between the current machine state and the available jobs. In this setting, a small MLP may be sufficient to calculate useful similarity scores.

The dropout also had a clear effect. A dropout of 0.2 was too strong in the tested setup. It did not lead to better generalization but rather prevented the model from learning useful scheduling behavior. A dropout of 0.1, on the other hand, worked better. This indicates that some regularization is useful, but too much regularization hurts the learning process.

7.2.2 CNN Architectures

The tested CNN architectures did not converge well, as shown in section 6.1.2. The most likely explanation is that the input representation used in this thesis does not contain a meaningful spatial structure. CNNs are especially useful when local neighborhoods in the input are meaningful. This is the case for images, time series, or grid-like structures. In the representation used in this thesis, however, the order of the features and embeddings does not necessarily create such local neighborhoods.

Therefore, the poor performance of the CNN architectures should be interpreted as representation-dependent. It does not mean that CNNs are generally unsuitable for solving the JSSP. It means that, for the specific Graph-based representation used in this thesis, Convolutional filters did not provide an advantage. Future work should only examine CNNs further if the input representation is changed in a way that creates meaningful spatial or sequential patterns.

7.2.3 Comparison between MLPs and CNNs

The comparison between MLPs and CNNs in section 6.1.3 suggests that the choice of architecture must be aligned with the state representation. In this work, the Graph embedding already provided a structured representation of the JSSP. The remaining task was therefore closer to a compatibility-scoring problem than to a spatial pattern-recognition problem. This explains why the MLP performed better than the CNN.

The main conclusion for RQ1 is therefore not that one neural network type is universally better than the other. Rather, the conclusion is that the tested Graph-based MARL architecture benefits from a small and stable downstream network. In this setup, increasing the depth of the neural network or using a CNN did not improve convergence.

7.3 Analysis of RQ2: Learnability of Dynamic Factors

The goal of RQ2 was to analyze how many of the introduced dynamic factors can be learned by Graph-based MARL Scheduling algorithms. The two dynamic factors tested in this thesis were machine delays and job delays. The results show that these two dynamic factors differ substantially in terms of learnability.

7.3.1 Machine Delays

Machine delays produced the strongest learnability signal. The dynamic policy consistently improved over the static policy and achieved the best result in most machine-delay configurations. This indicates that the algorithm was able to adapt its scheduling behavior to the dynamic behavior of the machines.

One reason for this result is that machine delays are connected to fixed resources. If a machine repeatedly creates delays, this behavior remains tied to the same machine. The agent can therefore learn that assigning jobs to this machine may be less beneficial if alternative machines are available. These patterns can be represented in the machine state and can be used by the agent to improve its decisions.

This result is relevant for practical applications because machine-related disturbances are common in real Shopfloors. Examples include slower machines, machine wear, planned or unplanned maintenance, and machine-specific processing uncertainty. The results suggest that Graph-based MARL can learn such disturbances if they are represented clearly enough in the state.

7.3.2 Job Delays

Job delays were more difficult to learn than machine delays. As shown in section 6.2.2, the dynamic policy consistently but only slightly outperformed the static policy in the scenario medians. However, this improvement was much smaller than for machine delays, and the heuristic baselines achieved the best makespan in several configurations.

The most likely explanation is that job delays are moving disturbances. A delayed job moves through the Shopfloor and affects different machines depending on its position in the schedule. Its effect also depends on predecessor jobs, successor jobs, machine availability, and the set of alternative machines. Therefore, the disturbance is not tied to one fixed resource. This makes it harder for the algorithm to identify stable patterns.

This does not necessarily mean that job delays cannot be learned by Graph-based MARL. Rather, the results indicate that the current feature set and architecture were not sufficient to learn job delays reliably. More informative job-related features may be necessary. Examples include delay history, remaining processing time, successor-job information, or features describing the criticality of a job within an order.

7.3.3 Why Both Dynamic Factors Were Not Combined

As discussed in section 6.2.4, both dynamic factors were not combined in this thesis. This was a methodological decision. If machine delays and job delays had been combined, the resulting experiment would have been difficult to interpret. If the algorithm achieved good results, it would not be clear whether this was caused by learning both factors or mainly by learning machine delays. If the algorithm achieved bad results, it would not be clear whether job delays were too difficult to learn or whether the combination of both factors made the environment too stochastic.

The author therefore argues that combining both factors is not reasonable in this thesis. RQ2.2 can only be answered after both dynamic factors have been shown to be learnable individually. Since the job-delay signal was weak and less robust than the machine-delay signal, no further combined analysis was conducted. A combined experiment would not have allowed a clear attribution of the observed behavior to one or both dynamic factors. Future work should first improve the representation and learning behavior for job delays before both dynamic factors are combined.

7.4 Methodological Implications of Order Generation on the JSSP

During the execution of the experiments, it was shown that the way the orders are generated has a heavy impact on what the algorithm is actually trying to solve. This section provides an overview of the lessons learned.

The central insight is that the generated instance does not only define the test case; it defines what can be learned. If the generated orders do not allow enough room for improvement, then even a good algorithm cannot improve the makespan by much. On the other hand, if the generated orders create bottlenecks that are too dominant or too random, the algorithm may also not be able to learn useful behavior. Therefore, order generation is not only a technical part of the experimental setup, but a central methodological factor in the evaluation of dynamic and flexible JSSP algorithms.

7.4.1 Schedules Need to Have Room for Improvement

The utilization of the machines should not start at a very high level. If there are many orders and the instance is fully flexible, then the utilization is almost 100% regardless of the algorithm, because there is always a job that can be executed when a machine is IDLE. In such a case, the makespan can be roughly calculated beforehand by dividing the total duration of all jobs by the number of machines. Furthermore, if there are jobs or machines that are constantly delayed, the algorithm cannot consider this behavior such that the makespan is not increased.

This means that the schedule of such orders without room for improvement are almost optimal by default. The optimization problem is then reduced to the task of picking jobs such that all machines finish their last job at approximately the same time. This problem is not very reasonable from a theoretical point of view because, if there is no room for improvement, the algorithm cannot learn anything meaningful. It does not make sense from a practical point of view: if a Shopfloor is already at full capacity, the reasonable choice to increase throughput is an increase in Shopfloor capacity, not smarter scheduling.

This leads to the insight that the orders and jobs must be designed such that the makespan and utilization can be improved. The following factors are important to consider:

- The number of orders cannot be too high; too many orders lead to an instance where there is always a job that can be executed on a machine.
- The number of orders cannot be too low if the instance is not flexible; if each job can only be executed on one machine and the number of orders is too low in the given context, then the makespan is mostly defined by the order with the longest total duration of all jobs.

These factors further lead to the insight that trying to solve the dynamic and/or flexible JSSP in general is very difficult. If and how a smart algorithm is able to improve the makespan is highly case-specific. As this thesis aimed to view this problem from a general perspective, this was a major challenge while designing the experimental pipeline.

7.4.2 Designing Reasonable Due Dates Is Difficult to Generalize

During the conduction of the experiments, it became clear that designing due dates for the orders is extremely difficult. For such a feature to be relevant, the following must hold:

- Each due date must be at least as far in the future as the sum of the job durations. This can be denoted as a lower bound, which is easy to calculate.
- However, there are multiple orders in parallel. This means that the actual lower bound must take into account all other orders, i.e., their respective total job durations and on which machines those jobs could be executed. If the JSSP instance is flexible, this increases the search space for finding a reasonable due date. The determination of reasonable due dates therefore results in a combinatorial search space.
- If the due date is surpassed, the effect of this violation must be defined, i.e. a penalty which may also differ across different orders.

These issues may be solvable for a specific set of orders by hand, but it is difficult to generalize them. In total, thousands of different orders were used. If due dates should be used, it is necessary to create a separate algorithm to solve these problems. This was deemed out of scope, and due dates were therefore not used.

7.4.3 Why Order Cancellation Cannot Be Learned without Extending the Problem Statement

At the beginning of this thesis, it was planned to also test algorithm behavior when orders are canceled during runtime. This is also the reason why fields such as `was_canceled` and `cancellation_time` are present in figure 4.3. However, it became clear that such

behavior cannot be reasonably learned by an algorithm without further information, such as why the order was canceled.

One theoretical scenario where this would make sense is if information about customers with noticeable cancellation behavior is available. In that case, the algorithm could potentially learn that certain orders have a higher probability of being canceled. Such scenarios were deemed out of scope for this work, and those features were not used.

7.5 Offline vs. online learning

One decision made in this work was to use Offline-RL instead of Online-RL for RQ2. Both variants have benefits and drawbacks in the context of the dynamic JSSP. In real production environments, if a dynamic algorithm is deployed, it will be necessary to retrain the algorithm.

On the one hand, online learning provides the option to continuously learn while executing jobs. This can have the benefit of continuously adjusting the model to the current state of the Shopfloor. This may be useful in environments where the behavior of the Shopfloor changes frequently and where the learning process itself can be safely integrated into production.

On the other hand, offline learning can be retrained at specific intervals, for example, every month or week. In the context of Shopfloor Scheduling, this approach allows for much more control in a real-life scenario:

- There may be a period of production that is a clear outlier and should not be included in the training. With offline learning, it is much easier to control which training data should be included after a human in the loop decides which data is relevant for the algorithm and which is not.
- In the case of a shift in the algorithm's performance, offline learning allows for easier comparisons due to its learning capability stemming from a replay buffer. Retrospectively, different algorithms can be trained and compared against each other more easily than with online learning.
- offline learning also allows for safer deployment because a model can be evaluated before it is used in the actual Shopfloor environment.

Overall, for the setup of this thesis, the benefits of using offline learning were considered more relevant than the benefits of online learning. With this argument, online learning was not included in this work. However, this should be interpreted as a design rationale and not as experimental evidence that offline learning is generally superior to online learning for the dynamic and flexible JSSP.

7.6 Limitations and Threats to Validity

This section summarizes the main limitations of this work. The limitations are grouped into construct validity, internal validity, conclusion validity, and external validity.

7.6.1 Construct Validity

The first limitation regarding construct validity is the lack of different objective functions. The objective function tested in this master thesis was implemented according to Park et al. [13]:

$$\overline{M}(\pi_\theta, \pi_b) = \frac{M(\pi_\theta) - M(\pi_b)}{M(\pi_b)} \tag{7.1}$$

I.e., this is a normalized makespan as described in section 3.1.

Other objective functions have not been tested. As already argued in section 3.4, the literature posits that makespan is the most important objective function to be optimized, as long as other exotic factors, such as energy consumption, are not factored in. However, other works did use different objective functions and compared the makespan afterward. Examples of such objectives can be found in section 2.3.6. This approach is not covered in this work.

A second limitation is that the makespan and utilization were the main metrics used for evaluation. As discussed in section 7.1.1, both metrics were strongly correlated in many experiments, but they do not capture all relevant aspects of a real production environment. Real Shopfloors may also optimize tardiness, robustness, energy consumption, waiting time, or due-date violations.

7.6.2 Internal Validity

The first limitation regarding internal validity is the fixed Feature Set. It was deemed out of scope for this work to test different feature sets. As this feature set was used for both RQ1 and RQ2, an improved representation of the machine state may have a major impact on the best hyperparameters and performance.

This limitation is especially relevant for job delays. The results indicate that the current feature representation may not represent moving disturbances well enough. Therefore, the inability to learn job delays in a stable way should not be interpreted as a general impossibility. It may instead be caused by the specific representation used in this thesis.

The second limitation is the lack of feature-importance analysis. The weights of the neural network were not analyzed with regard to feature importance, as explainability was deemed out of scope for this work. This may, however, be an interesting area for future work. As the features were mostly aligned with the literature, it is unclear which features led to a learning effect in the algorithm.

The third limitation is that hyperparameter tuning was only conducted on static JSSP instances. The setup of RQ1 and RQ2 indicated that the hyperparameters used for RQ2

were optimal for a static JSSP instance. It is unclear whether those parameters were also optimal for the dynamic JSSP; it is possible that different hyperparameters would lead to better results.

7.6.3 Conclusion Validity

A further limitation is that the main result tables primarily analyze median makespan and median utilization. While the appendix provides boxplots over the random seeds, the variance for specific instances is not systematically analyzed and interpreted to increase explainability.

This is especially relevant for RQ2, because the experiments include stochastic dynamic factors. The conclusions are strongest where the distributions are clearly separated and weaker where the distributions overlap. A more systematic analysis of variance, confidence intervals, and statistical significance would strengthen the interpretation of the results.

7.6.4 External Validity and Difficult Comparisons across the Literature

Another limitation regarding external validity is the choice of heuristic baselines. The heuristic baselines used in this thesis are intentionally simple. FIFO, SPT, and Random provide interpretable reference points and are useful for comparing the learned policies against basic rule-based behavior. However, they do not cover stronger dispatching rules, meta-heuristics, or optimization-based baselines. Therefore, outperforming these baselines should not be interpreted as outperforming the broader class of heuristic or optimization-based Scheduling algorithms.

The last few chapters showed how great the possibilities are when designing a flexible and dynamic JSSP, and how even small changes in the setup can have a major impact on performance, on how and whether the instances can be learned, and even on what is to be learned at all.

The author of this work argues that the combinatorial space of this problem and how the setup can be done are too great to be generalized easily. The search space would be greatly reduced if the work were based on or related to a more specific scenario; for example, a real situation where the behavior of the Shopfloor is more clearly defined.

This also limits the external validity of the results. The experiments were conducted on generated instances. These instances were necessary to evaluate the research questions in a controlled way, but they cannot capture all aspects of real production environments. Therefore, the results should be interpreted as evidence for the tested simulation setup and not as a direct guarantee for real Shopfloor deployment.

7.7 Future Work

Several directions for future work arise from this thesis.

First, future work should analyze different feature sets. In this thesis, the feature set was defined at the beginning of the work and was then used for both RQ1 and RQ2, as discussed in section 7.6. It was deemed out of scope to test multiple feature sets. However, the results indicate that the feature set may have a major impact on learning behavior. This is especially relevant for job delays. It is possible that the current feature set does not represent job delays well enough.

Second, the importance of the features used should be analyzed. The weights of the neural network were not analyzed in this thesis. Therefore, it is unclear which features were actually important for the algorithm's decision. Such an analysis could help to better understand why machine delays were learned while job delays were not learned in a stable way.

Third, future work should tune the hyperparameters directly on dynamic JSSP instances. In this thesis, the best architecture from RQ1 was used for RQ2. This is a reasonable approach because RQ1 analyzed the static flexible JSSP first. However, it is unclear whether the best hyperparameters for the static JSSP are also the best hyperparameters for the dynamic JSSP. Dynamic environments may need different learning rates, dropout values, replay buffer settings, or network sizes.

Fourth, job delays should be analyzed in more detail. The results of this thesis show that job delays are more difficult to learn than machine delays, as discussed in section 6.2.3. This may be caused by the fact that job delays are moving disturbances. A delayed job moves through the Shopfloor and affects different machines depending on its position in the schedule. Future work should therefore test additional features for job delays, such as delay history, remaining processing time, successor jobs, or information about the criticality of a job.

Fifth, the results for CNN architectures should be examined further only if the input representation is changed. In this thesis, CNN architectures did not converge well, as shown in section 6.1.2. The author argues that this is probably caused by the fact that the input representation does not have a meaningful spatial structure. Therefore, future work could either use CNNs with a different representation or focus on other architectures, such as attention-based networks.

Sixth, different objective functions should be tested. This thesis focused on makespan and utilization. Makespan is the most common objective function in JSSP literature, but real production environments may have additional goals. Examples include tardiness, waiting time, energy consumption, and robustness, as described in section 2.3.6. Future work could test whether the learning behavior changes when other objective functions are used.

Seventh, future work should include a more systematic statistical analysis of the stochastic experiments. The appendix already provides boxplots over the random seeds, but

future work could analyze variance, confidence intervals, and statistical significance more explicitly. This would make it easier to distinguish stable learning effects from random variation.

Finally, future work should test the approach on real or semi-real production data. The results of this thesis show that the generated instances have a strong impact on the learning behavior, as discussed in section 7.4. If the generated orders do not create enough bad decisions, the algorithm cannot improve the makespan by much. Real production data could help analyze whether the algorithm also works in a more realistic Shopfloor setup.

CHAPTER 8

Conclusion

This thesis investigated Graph-based MARL for the dynamic and flexible JSSP. The motivation was the need for adaptive and scalable Scheduling algorithms in Smart Manufacturing, where Shopfloors are affected by flexibility, changing requirements, and uncertainty.

The work addressed two main questions. First, it analyzed how the neural network architecture of the agents affects convergence on the flexible JSSP. Second, it investigated whether different dynamic factors can be learned by a Graph-based MARL Scheduling algorithm. For this purpose, a Shopfloor environment was implemented that can generate static, flexible, and dynamic JSSP instances. The algorithm used a Graph-based representation of jobs and machines, and the resulting embeddings were used by the agents to select jobs for machines.

For RQ1, the results show that MLP architectures were more suitable than CNN architectures in the tested setup. The best performance was achieved by a small MLP with two layers and a dropout of 0.1. Larger networks did not improve the results. This suggests that the Graph embedding already captures much of the problem structure, so the agent mainly has to learn a scoring function between the current machine state and the available jobs. The tested CNN architectures were less suitable because the input representation does not provide a meaningful spatial structure.

For RQ2, two dynamic factors were analyzed: machine delays and job delays. Machine delays produced the stronger learnability signal. The dynamic policy improved over the static policy in all machine-delay scenario medians and achieved the best makespan in most machine-delay configurations.

Job delays produced a weaker but still consistent improvement over the static policy. However, the magnitude of this improvement was small, and heuristic baselines achieved the best makespan in several job-delay configurations. Therefore, job delays should not be

interpreted as completely unlearned, but as less actionable and less robustly exploitable than machine delays in the tested setup.

For this reason, the simultaneous learning of both dynamic factors was not analyzed further. Since the job-delay signal was weak and less robust than the machine-delay signal, a combined experiment would not have allowed a clear attribution of the observed behavior to one or both dynamic factors.

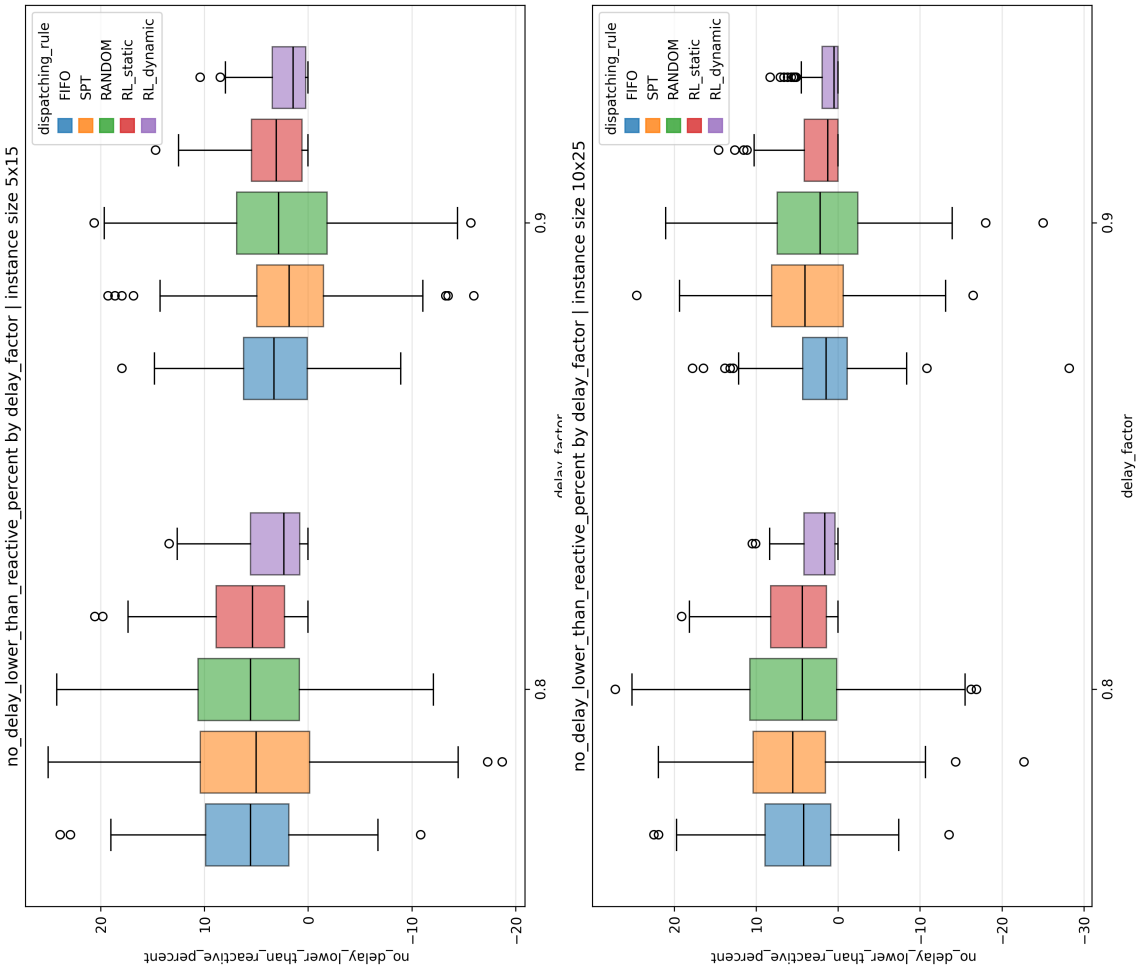
Machine delays and job delays differ substantially in their learnability. The current feature representation was sufficient for machine delays, but not for job delays. For this reason, the simultaneous learning of both dynamic factors was not analyzed further, as the unclear job-delay results would have made the combined results difficult to interpret.

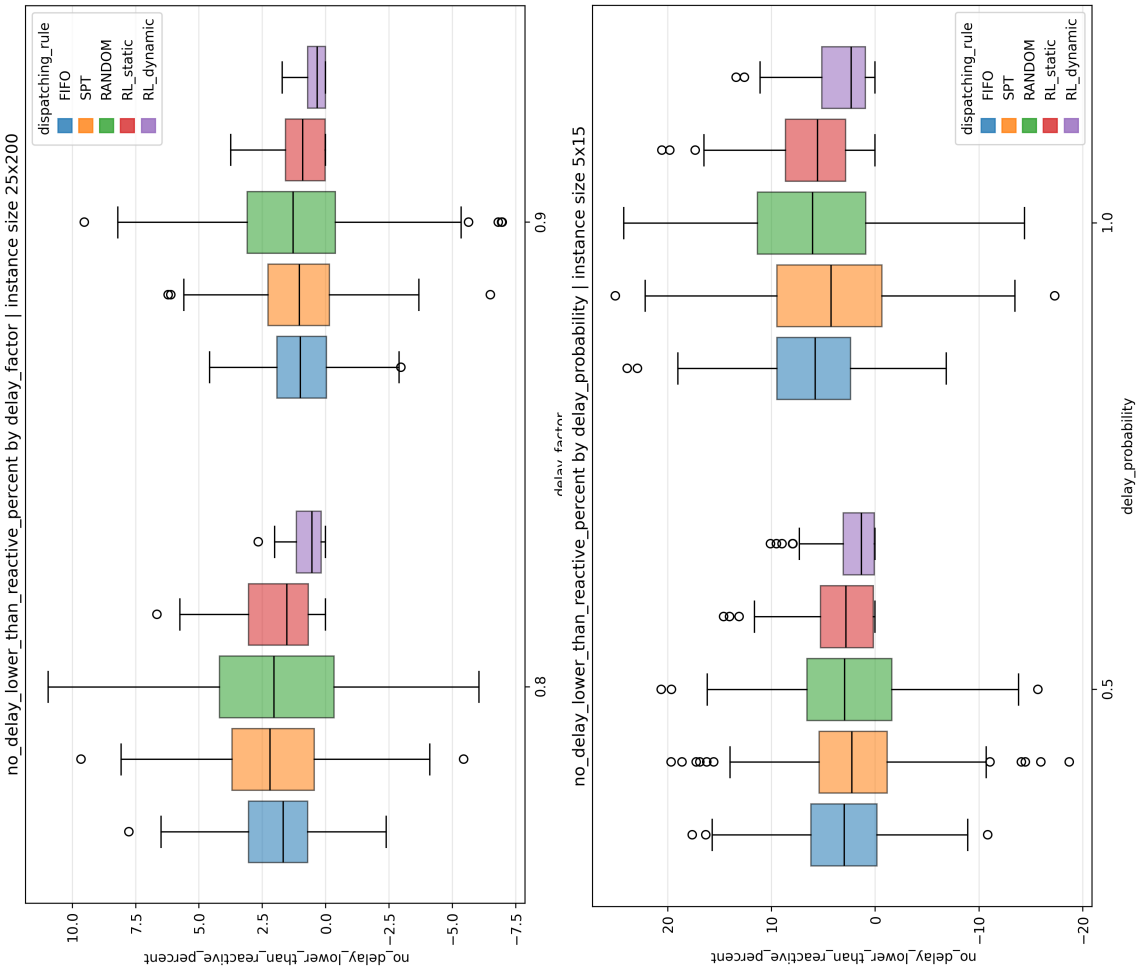
Overall, this thesis shows that Graph-based MARL is a reasonable approach for the dynamic and flexible JSSP, but its success depends strongly on the state representation and the type of uncertainty. The scientific contribution is the analysis of neural network architectures within a Graph-based MARL Scheduling algorithm and the differentiation between dynamic factors. The practical contribution is the insight that dynamic Scheduling algorithms must be evaluated with regard to the specific uncertainty present in the Shopfloor.

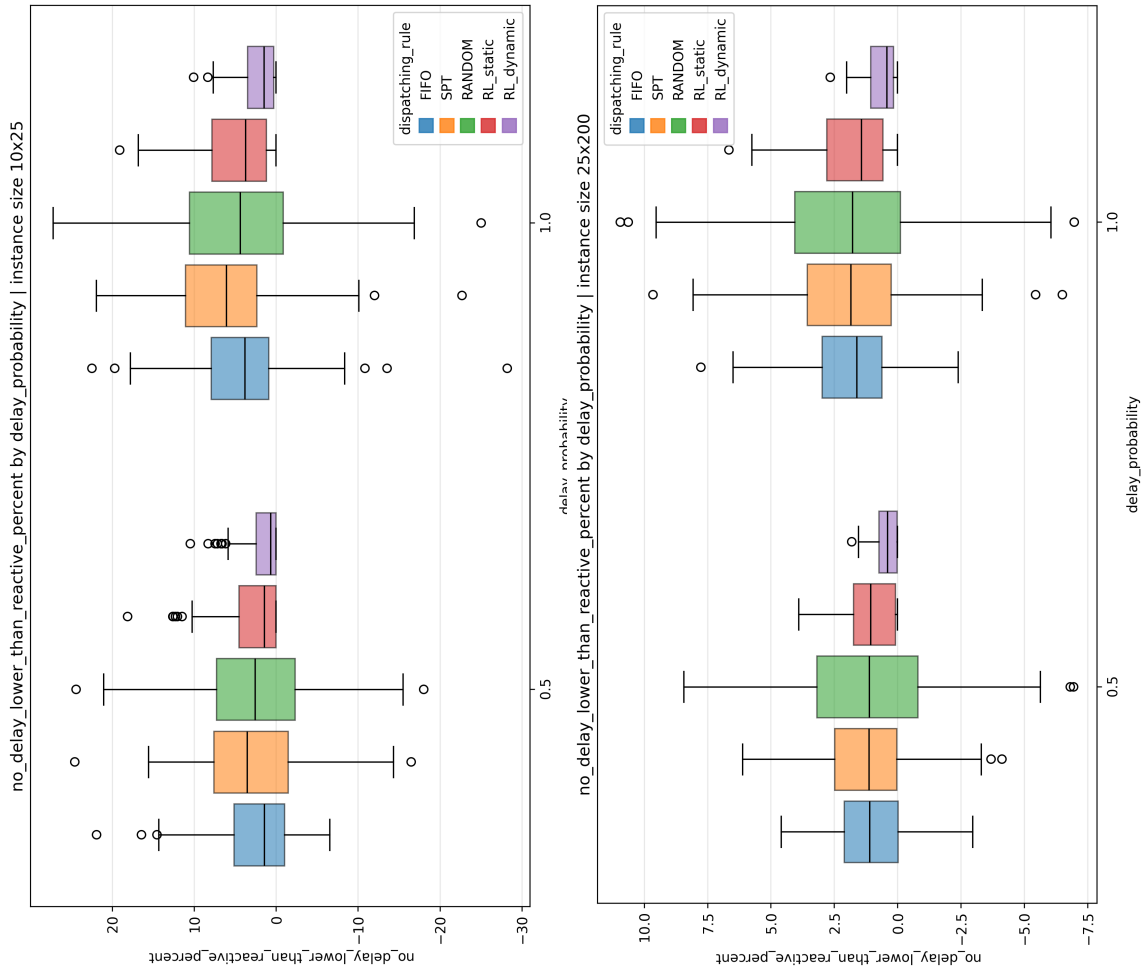
Several limitations remain. The feature set was fixed, feature importance was not analyzed, and the experiments were conducted on generated instances rather than real production data. Future work should therefore analyze different feature sets, especially for job delays, tune the architecture directly on dynamic instances, and test the approach on real Shopfloor data.

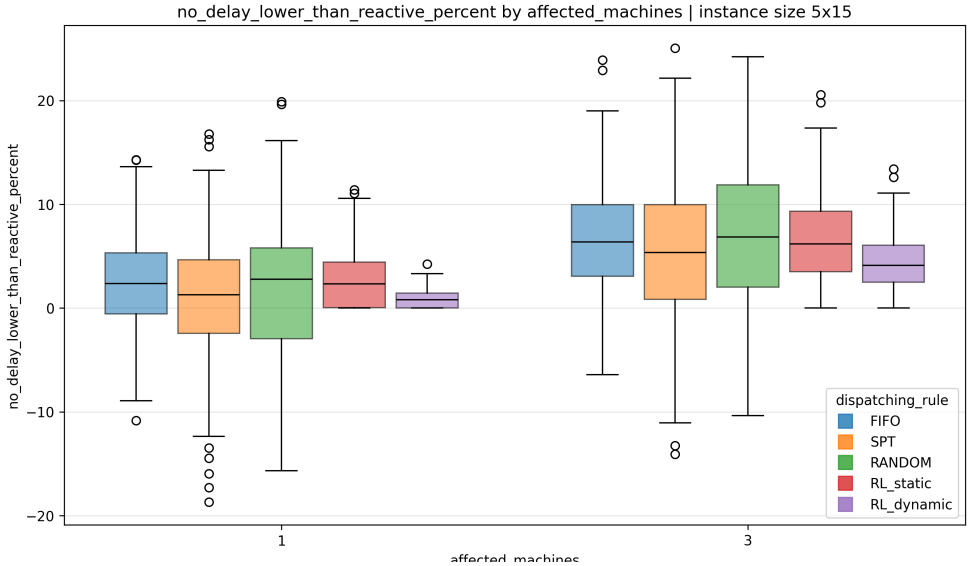
APPENDIX A

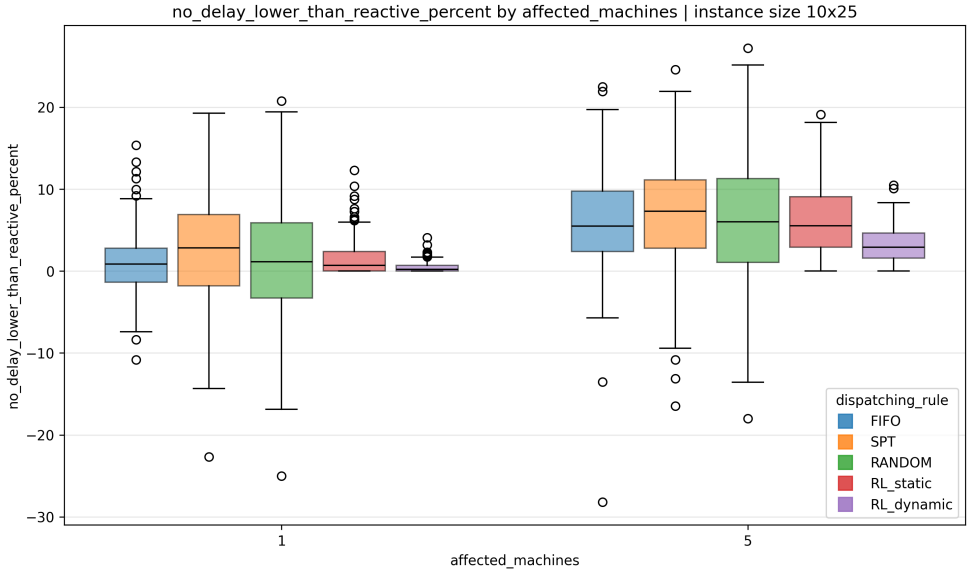
Boxplots for Machine Delays


no_delay_lower_than_reactive_percent by delay_factor | instance size 5x15
no_delay_lower_than_reactive_percent by delay_factor | instance size 10x25
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
delay_factor
0.8
0.9
20
10
0
−10
−20
−30


no_delay_lower_than_reactive_percent by delay_factor | instance size 25x200
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
0.8
0.9
delay_factor
no_delay_lower_than_reactive_percent by delay_probability | instance size 5x15
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
20
10
0
−10
−20
0.5
1.0
delay_probability

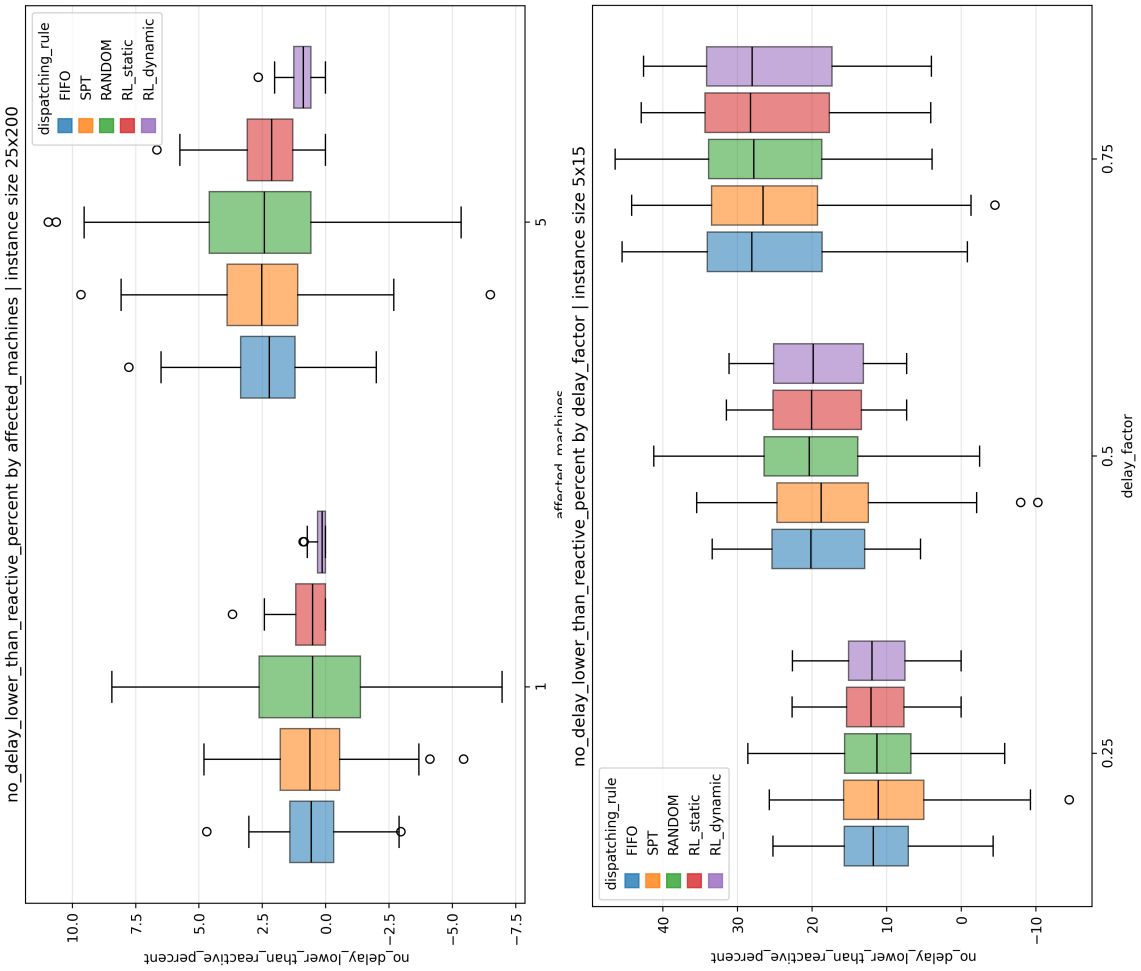

no_delay_lower_than_reactive_percent by delay_probability | instance size 10x25
no_delay_lower_than_reactive_percent by delay_probability | instance size 25x200
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
delay_probability

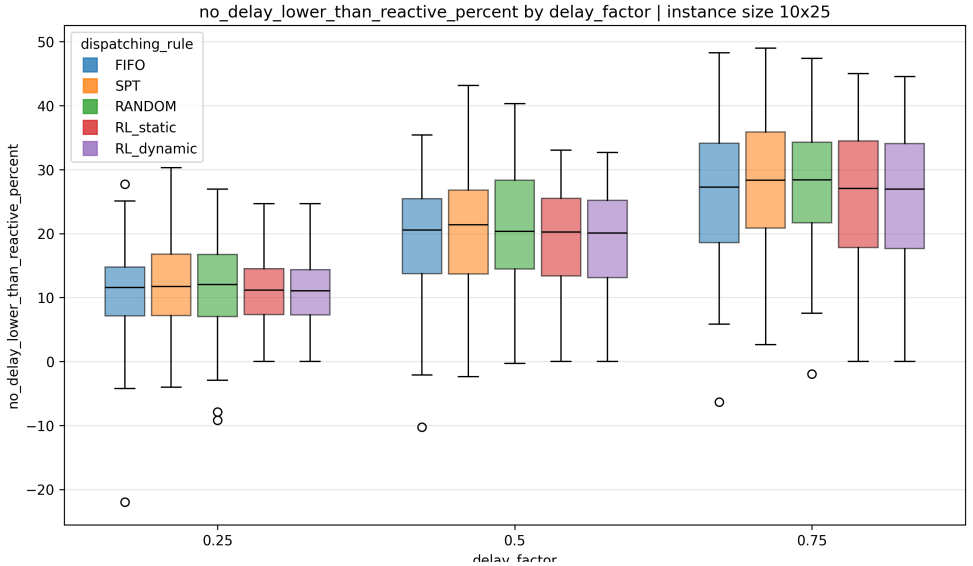

no_delay_lower_than_reactive_percent by affected_machines | instance size 5x15
no_delay_lower_than_reactive_percent
20
10
0
−10
−20
1
3
affected_machines
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic

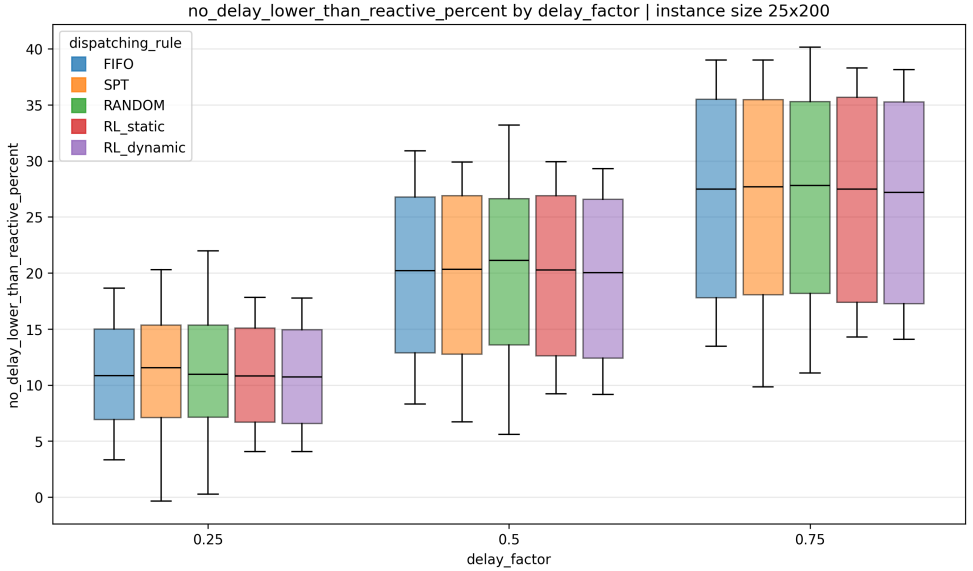

no_delay_lower_than_reactive_percent by affected_machines | instance size 10x25
no_delay_lower_than_reactive_percent
20
10
0
−10
−20
−30
1
5
affected_machines
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic

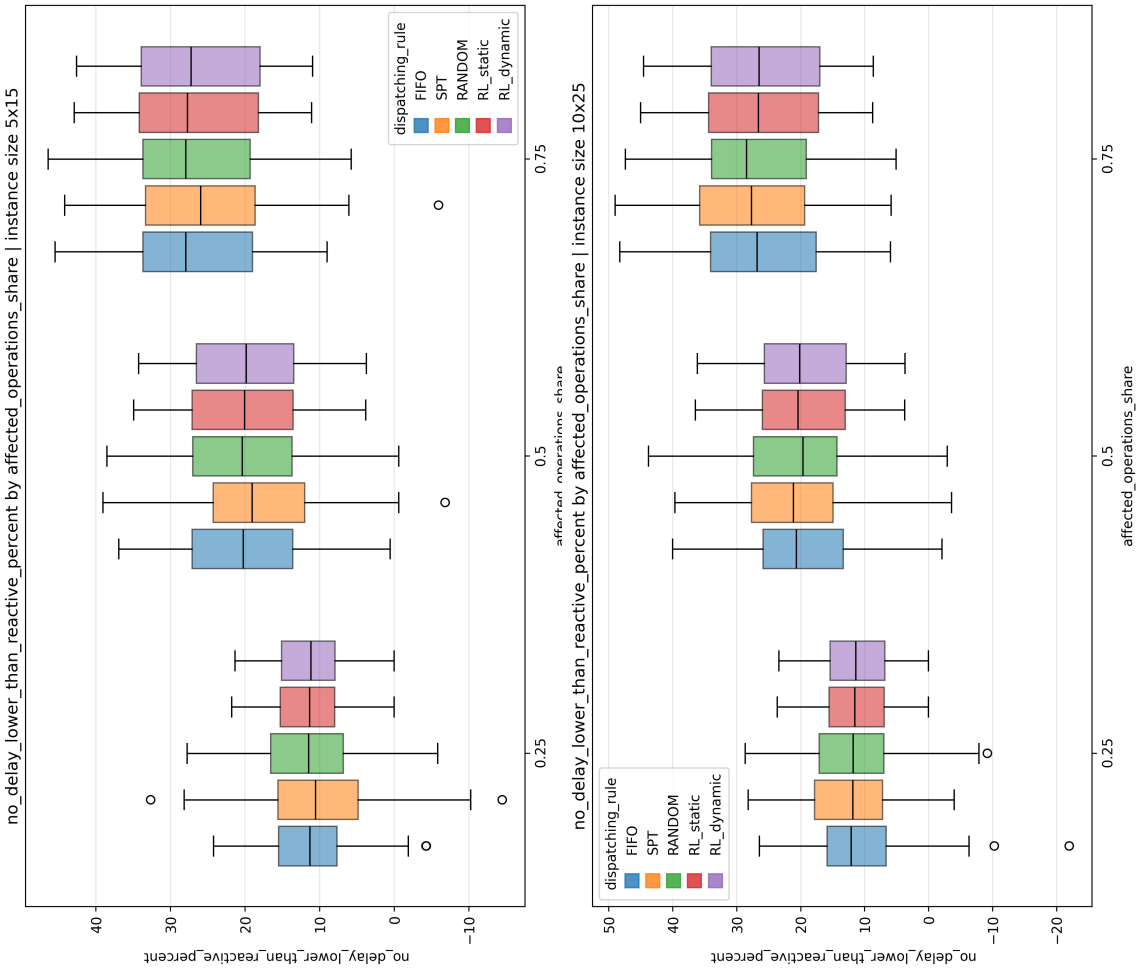
APPENDIX B

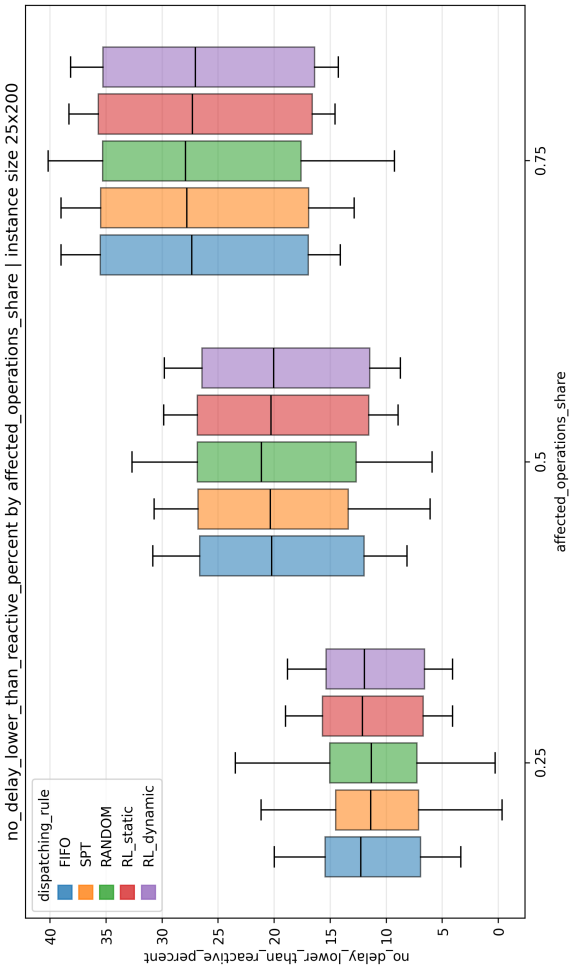
Boxplots for Job Delays


no_delay_lower_than_reactive_percent by affected_machines | instance size 25x200
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
affected_machines
no_delay_lower_than_reactive_percent by delay_factor | instance size 5x15
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
delay_factor


no_delay_lower_than_reactive_percent by delay_factor | instance size 10x25
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
50
40
30
20
10
0
−10
−20
0.25
0.5
0.75
delay_factor


no_delay_lower_than_reactive_percent by delay_factor | instance size 25x200
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
40
35
30
25
20
15
10
5
0
0.25
0.5
0.75
delay_factor


no_delay_lower_than_reactive_percent by affected_operations_share | instance size 5x15
no_delay_lower_than_reactive_percent by affected_operations_share | instance size 10x25
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
affected_operations_share


no_delay_lower_than_reactive_percent by affected_operations_share | instance size 25x200
dispatching_rule
FIFO
SPT
RANDOM
RL_static
RL_dynamic
no_delay_lower_than_reactive_percent
0
5
10
15
20
25
30
35
40
affected_operations_share
0.25
0.5
0.75

Overview of Generative AI Tools Used

Programming

The author used the 4o-model from OpenAI in various ways to support the code writing. With the exceptions listed below ('Prompts Used To Create Whole Functionalities'), the author never copied full functionalities from the chat.

Prompts to (Further) Ensure a Proper Architecture

Different types of prompts have been used to provide some kind of knowledge before the implementation of various steps:

- Explain me how Subject X works in theory
- Suggest various different ways on how problem X can solved based on the literature (use DeepResearch option).
- Explain different approaches of implementations for Task X in Python

However, these prompts were not directly used to gather knowledge about the topic (the author already read at least 90% of the literature referenced in this document, and was already very well aware of most coding principles and available frameworks). The author was well aware of the architecture and needed functionalities before the actual implementation. Merely, the prompts have been used as an addition to the authors initial knowledge and research as an additional 'safety net' to not oversee important aspects.

Prompts Used for Debugging

Generative AI is, at least for now, not able to detect logical errors over 15.000 lines of code. However, it can be of great help for various simple syntax issues. Those code snippets of maximal 3 lines have been either copied directly (as it was a standalone issue, such as the prompt 'how to convert a (5,) Tensor to a (1,5) Tensor in PyTorch', potentially followed up by questions such as 'explain function X in general'), or directed

the author into the correct direction of fixing the error (On the prompt 'why do I get error X', answers such as (paraphrased) 'because .keys() does not return a list of keys, you need to convert it to a list yourself')

Prompts Used to Create Whole Functionalities

Shell scripts used for deployment and automation were drafted with the support of generative AI and then reviewed, adapted, and tested by the author.

Writing

Grammarly

Grammarly was used in the last phase of writing to check for grammatical errors, bad language, or similar. The suggested text was not directly copied back, the author decided for each suggestion separately if it is deemed as a relevant change.

Simple Prompts Regarding TeX-Files

The 4o-model from OpenAI was used to ask simple questions about writing tex-files in overleaf. The list of prompts is as follows:

- How to add links with text in overleaf
- How can I make my research questions 'clickable' in overleaf?
- Associate Prof. Dr.techn. Dipl.-Ing.: Which of these title belong before/after the name in a master's thesis?

Formulas and Algorithms

Various questions of how to create various mathematical notations for the formulas in LaTeX have been asked, such as 'How do I make a horizontal line over a capital letter in latek math?'.
Some of the complex formulas have been created by providing a screenshot of the formula/algorithm from the literature together with the Prompt ' Print this formula such that I can copy it into my latek-document'. All formulas and algorithms which have been created this way were additionally checked if the model indeed created an exact replication of the literature.

Sentences Used

The following text has been generated: 'While average-reward models can simplify the development of decentralized MARL algorithms, they often require explicit communication protocols between agents. Designing such protocols is highly context-dependent,

and attempts to generalize them invariably confront the deeper, unresolved problem of linguistic formalization. In essence, enabling generalized communication between agents raises the same fundamental questions that underlie human language: what constitutes meaning, and how is it effectively conveyed?'
This was created with the prompt: 'Average-reward models ease the development of decentralized MARL algorithms, but often necessitates the incorporation of communication protocols into MARL. Designing a good communication protocol is often very case-specific. Attempts to generalization lead back to the formalization of linguistics, which is unsolved in many facets. The point I want to make is that generalized communication between the agents lead back to the issue of what is linguistics and how do humans communicate effectively. 1. rate this argument 2. make a better formulation'

Correction

The Chat GPT Model 5.5 with extended thinking was used to analyze the whole document and make suggestions for inconsistent or incomplete argumentation and check for under- or overemphasized aspects.

Acronyms

A3C	Asynchronous Advantage Actor-Critic Algorithm
AI	Artificial Intelligence
APS	Advanced Planning and Scheduling
CNN	Convolutional Neural Network
DPR	Dispatching Rules
DQN	Deep Q-Networks
DRL	Deep Reinforcement Learning
EFG	Extensive-Form Games
ERP	Enterprise Resource Planning
FFN	Feed-Forward Neural Network
FIFO	First-In-First-Out
GAT	Graph Attention Network
GNN	Graph Neural Network
GPI	Generalized Policy Iteration
GPU	Graphics Processing Units
JSSP	Job-Shop Scheduling Problem
KPI	Key Performance Indicators
LPTR	Longest-Processing-Time-Remaining
MARL	Multi-Agent Reinforcement Learning

MC	Monte Carlo Algorithm
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
MES	Manufacturing Execution Systems
MG	Markov Game
ML	Machine Learning
MLP	Multi-Layer Perceptron
NE	Nash Equilibrium
NN	Neural Network
PPC	Production Planning and Control
PPO	Proximal Policy Optimization Algorithm
RL	Reinforcement Learning
SARSA	State-Action-Reward-State-Action Algorithm
SOTA	State-Of-The-Art
SPT	Shortest-Processing-Time
TD	Temporal Difference Algorithm
TRPO	Trust Region Policy Optimization Algorithm
TSP	Traveling-Salesman Problem

List of Symbols

One JSSP-Instance:

$\mathcal{O}$	Set of Orders
$\mathcal{M}$	Set of Machines
J_{oi}	Job J from Order o at position i
p_{oi}	Processing Time of Job J_{oi}
s_{oi}	Start Time of J_{oi}
c_{oi}	Completion Time of J_{oi}
m	Machine $m \in \mathcal{M}$
m_{oi}	The subset of Machines on which J_{oi} can be processed
c_{max}	Makespan
σ	Schedule
σ^*	optimal Schedule

Reinforcement Learning:

s, s'	States
a	An action
r	A reward
$\mathcal{S}$	Set of all non-terminal states
$\mathcal{S}^+$	Set of all states, including the terminal state
$\mathcal{A}(s)$	Set of all actions available in state s
$\mathcal{R}$	Set of all possible rewards, a finite subset of $\mathbb{R}$
t	Discrete time step
a_t	Action at time t
s_t	State at time t
r_t	Reward at time t
π	Policy (decision-making rule)
$\pi(s)$	Action taken in state s under deterministic policy π
$\pi(a\|s)$	Probability of taking action a in state s under stochastic policy π
G_t	Return following time t

$\boldsymbol{p(s'\|s,a)}$	Probability of transitioning to state s' when taking action a in state s
$\boldsymbol{v_\pi(s)}$	Value of state s under policy π (expected return)
$\boldsymbol{v_*(s)}$	Value of state s under the policy
$\boldsymbol{q_\pi(s,a)}$	Value of taking action a in state s under policy π
$\boldsymbol{q_*(s,a)}$	Value of taking action a in state s under the optimal policy
$\boldsymbol{V}$,$\boldsymbol{V_t}$	Array estimates of state-value function v_π or v_*
$\boldsymbol{V_t(s)}$	Estimate of $v_*(s)$ at timestep t
$\boldsymbol{Q}$,$\boldsymbol{Q_t}$	Array estimates of action-value function q_π or q_*
$\boldsymbol{Q_t(a)}$	Estimate of $q_*(a)$ at timestep t
$\boldsymbol{\epsilon}$	Probability of taking a random action in an ϵ-greedy policy
$\boldsymbol{\alpha, \beta}$	Step-size parameters
$\boldsymbol{\gamma}$	Discount rate
$\boldsymbol{\lambda}$	Decay-rate parameter for eligibility traces
Neural Networks:	
$\boldsymbol{\theta}$	NN Parameters
$\boldsymbol{\delta}$	Gradient
$\boldsymbol{q(s,a;\theta)}$	Value of taking action a in state s under the parameters θ
$\boldsymbol{A_t}$	Advantage at time t
$\boldsymbol{\hat{A}_t}$	Estimated Advantage at time t
$\boldsymbol{\epsilon}$	Clipping paramater used in PPO
Multi-Agent RL:	
$\boldsymbol{\mathcal{N}}$	Set ofagents
$\boldsymbol{\mathcal{S}}$	State space observed by allagents
$\boldsymbol{A^i}$	Action space of agent i
$\boldsymbol{\mathcal{P}}$	transition probability from any state s to any state s' for any joint action a
$\boldsymbol{\mathcal{R}^i}$	Reward function determining the immediate reward for agent i
$\boldsymbol{a_t^i}$	Action of agent i at timestep t
$\boldsymbol{r_t^i}$	Reward of agent i at timestep t when performing action a in state s
$\boldsymbol{\pi^i}$	Policy of agent i
$\boldsymbol{V^i_{\pi^i,\pi^{-i}}(s)}$	Value function in state s for agent i on the joint policy π, where $-i$ denotes the other agents
$\boldsymbol{\pi^*}$	Optimal joined policy (Nash Equilibrium)
$\boldsymbol{\mathcal{H}}$	Set of all possible histories
$\boldsymbol{ha^i}$	History of agent i
$\boldsymbol{\mathcal{A}^i(h)}$	The action space of agent i given history h
$\boldsymbol{\mathcal{Z}}$	Subset of terminal histories the represent the completion of an EFG
$\boldsymbol{\tau}$	An identification function mapping agents to histories
$\boldsymbol{c}$	A special agent called 'chance' specifying the randomness of the environment

$\boldsymbol{S}$	A partition of H, i.e. a subset of histories
$\boldsymbol{\eta_\pi(h)}$	Reach probability for history h
Graphs:	
$\boldsymbol{V}$	Set of vertices/nodes
$\boldsymbol{\mathcal{E}}$	Set of edges
$\boldsymbol{u, v}$	Edges
$\boldsymbol{h_u}$	Vector representation of node u
$\boldsymbol{\mathcal{N}_u}$	Neighborhood of node u
$\boldsymbol{X_{\mathcal{N}_u}}$	Multi-set of all neighborhood features
$\oplus$	Any permutation-invariant aggregation function
$\boldsymbol{\phi}$, $\boldsymbol{\psi}$	Neural Networks

List of Figures

2.1 Timeline of industrial revolutions from Industry 1.0 to Industry 5.0 according to [17] . . . 9
2.2 Automation Pyramid . . . 10
2.3 MES functionalities according to [19] . . . 11
2.4 APS functionalities according to [20] . . . 12
2.5 Scheduling approaches according to [21] . . . 15
2.6 Interaction between agent and environment according to [25] . . . 20
2.7 A level-based foraging task with three robots [28] . . . 30
2.8 JSSP Instance Represented In A Graph [37] . . . 36
2.9 Mapping From Smart Manufacturing Features To MARL [40] . . . 38

4.1 Architecture Of The Shopfloor-Simulator . . . 51
4.2 Probabilistic state machine . . . 52
4.3 ERD of the Database . . . 54
4.4 ERD of the Postgres DB in Exoscale . . . 59
4.5 job Execution Pipeline . . . 63

List of Tables

5.1 Instance sizes used in the experiments. 78
5.2 Number of eligible machines per job induced by the 30% flexibility setting. 80
5.3 Shared training parameters. 82
5.4 MLP hyperparameter grid for RQ1. 82
5.5 CNN hyperparameter grid for RQ1. 83
5.6 machine-delay parameter grid for RQ2. 84
5.7 job-delay parameter grid for RQ2. 85

6.1 machine-delay reactive performance for size 5×15. Reported values are medians over 50 evaluation instances. 92
6.2 machine-delay reactive performance for size 10×25. Reported values are medians over 50 evaluation instances. 93
6.3 machine-delay reactive performance for size 25×200. Reported values are medians over 50 evaluation instances. 94
6.4 job-delay reactive performance for size 5×15. Reported values are medians over 50 evaluation instances. 97
6.5 job-delay reactive performance for size 10×25. Reported values are medians over 50 evaluation instances. 98
6.6 job-delay reactive performance for size 25×200. Reported values are medians over 50 evaluation instances. 99

List of Algorithms

2.1 SARSA (on-policy TD control) for estimating $Q \approx q_*$ according to [25] 24

2.2 Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$ according to [25] 25

2.3 Asynchronous one-step Q-learning: pseudo-code for each actor-learner thread based on [30] 27

Bibliography

[1] Behice Meltem Kayhan and Gokalp Yildiz. "Reinforcement learning applications to machine scheduling problems: A comprehensive literature review". In: *Journal of Intelligent Manufacturing* 34.3 (Mar. 2023), pp. 905–929. DOI: 10.1007/s10845-021-01847-3. URL: https://link.springer.com/10.1007/s10845-021-01847-3 (visited on 03/26/2025).

[2] Adeniran Afolalu et al. "The role of production planning in enhancing an efficient manufacturing system – an overview". In: *E3S Web of Conferences* 309 (Oct. 2021), p. 01002. DOI: 10.1051/e3sconf/202130901002.

[3] Jian Zhang et al. "Review of job shop scheduling research and its new perspectives under Industry 4.0". In: *Journal of Intelligent Manufacturing* 30.4 (Apr. 2019), pp. 1809–1830. DOI: 10.1007/s10845-017-1350-2. URL: https://doi.org/10.1007/s10845-017-1350-2 (visited on 04/08/2025).

[4] David Applegate and William Cook. "A computational study of the job-shop scheduling problem". In: *ORSA Journal on Computing* 3.2 (May 1991), pp. 149–156. DOI: 10.1287/ijoc.3.2.149. URL: https://pubsonline.informs.org/doi/abs/10.1287/ijoc.3.2.149 (visited on 04/08/2025).

[5] Gur Mosheiov. "Complexity analysis of job-shop scheduling with deteriorating jobs". In: *Discrete Applied Mathematics* 117.1 (Mar. 2002), pp. 195–209. DOI: 10.1016/S0166-218X(00)00385-1. URL: https://www.sciencedirect.com/science/article/pii/S0166218X00003851 (visited on 04/08/2025).

[6] Chathurangi Shyalika, Thushari Silva, and Asoka Karunananda. "Reinforcement learning in dynamic task scheduling: A review". In: *SN Computer Science* 1.6 (Nov. 2020), p. 306. DOI: 10.1007/s42979-020-00326-5. URL: https://link.springer.com/10.1007/s42979-020-00326-5 (visited on 03/26/2025).

[7] Wen Song et al. "Flexible job-shop scheduling via graph neural network and deep reinforcement learning". In: *IEEE Transactions on Industrial Informatics* 19.2 (Feb. 2023), pp. 1600–1610. DOI: 10.1109/TII.2022.3189725. URL: https://ieeexplore.ieee.org/abstract/document/9826438 (visited on 07/20/2025).

[8] Wei Zhang and Thomas G. Dietterich. “A reinforcement learning approach to job-shop scheduling”. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI’95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 1995, pp. 1114–1120. (Visited on 04/08/2025).

[9] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. July 2019. DOI: 10.48550/arXiv.1509.02971. URL: http://arxiv.org/abs/1509.02971 (visited on 03/26/2025).

[10] Todd Hester et al. “Deep Q-learning from demonstrations”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). Number: 1. DOI: 10.1609/aaai.v32i1.11757. URL: https://ojs.aaai.org/index.php/AAAI/article/view/11757 (visited on 04/08/2025).

[11] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. “Multi-agent reinforcement learning: A selective overview of theories and algorithms”. In: *Handbook of Reinforcement Learning and Control*. Ed. by Kyriakos G. Vamvoudakis et al. Vol. 325. Cham: Springer International Publishing, 2021, pp. 321–384. DOI: 10.1007/978-3-030-60990-0_12. URL: https://link.springer.com/10.1007/978-3-030-60990-0_12 (visited on 03/26/2025).

[12] Shiyong Wang et al. “Design patterns of deep reinforcement learning models for job shop scheduling problems”. In: *Journal of Intelligent Manufacturing* (July 2024). DOI: 10.1007/s10845-024-02454-8. URL: https://link.springer.com/10.1007/s10845-024-02454-8 (visited on 03/26/2025).

[13] Junyoung Park, Sanjar Bakhtiyar, and Jinkyoo Park. *ScheduleNet: Learn to solve multi-agent scheduling problems with reinforcement learning*. June 2021. DOI: 10.48550/arXiv.2106.03051. URL: http://arxiv.org/abs/2106.03051 (visited on 03/26/2025).

[14] Keyulu Xu et al. *How powerful are graph neural networks?* Feb. 2019. DOI: 10.48550/arXiv.1810.00826. URL: http://arxiv.org/abs/1810.00826 (visited on 03/26/2025).

[15] Petar Veličković et al. *Graph attention networks*. Feb. 2018. DOI: 10.48550/arXiv.1710.10903. URL: http://arxiv.org/abs/1710.10903 (visited on 07/20/2025).

[16] Peter Hines. “A categorical framework for finite state machines”. In: *Mathematical Structures in Computer Science* 13.3 (June 2003), pp. 451–480. DOI: 10.1017/S0960129503003931. URL: https://www.cambridge.org/core/journals/mathematical-structures-in-computer-science/article/categorical-framework-for-finite-state-machines/F604ECAC4395590C577602221608190D (visited on 04/08/2025).

[17] Francisco Javier Folgado et al. “Review of Industry 4.0 from the perspective of automation and supervision systems: Definitions, architectures and recent trends”. In: *Electronics* 13.4 (2024), p. 782.

[18] Olumide Emmanuel Oluyisola et al. “Designing and developing smart production planning and control systems in the Industry 4.0 era: A methodology and case study”. In: *Journal of Intelligent Manufacturing* 33.1 (2022), pp. 311–332.

[19] B. Saenz de Ugarte, A. Artiba, and R. Pellerin. “Manufacturing execution system – a literature review”. In: *Production Planning and Control* 20.6 (Sept. 2009), pp. 525–539. DOI: 10.1080/09537280902938613. URL: https://doi.org/10.1080/09537280902938613 (visited on 04/25/2025).

[20] Thales Botelho de Sousa et al. “An overview of the advanced planning and scheduling systems”. In: *Independent Journal of Management and Production* 5.4 (2014), pp. 1032–1049.

[21] Zengqiang Jiang et al. “The evolution of production scheduling from Industry 3.0 through Industry 4.0”. In: *International Journal of Production Research* 60.11 (June 2022), pp. 3534–3554. DOI: 10.1080/00207543.2021.1925772. URL: https://doi.org/10.1080/00207543.2021.1925772.

[22] António R Teixeira, José Vasconcelos Ferreira, and Ana Luísa Ramos. “Intelligent supply chain management: A systematic literature review on artificial intelligence contributions”. In: *Information* 16.5 (2025), p. 399.

[23] Alessandro Immordino et al. “Explainable AI for reinforcement learning based dynamic scheduling solutions in semiconductor manufacturing: A. Immordino et al.” In: *Journal of Intelligent Manufacturing* (2025), pp. 1–17.

[24] Shubhendu Kshitij Fuladi and Chang-Soo Kim. “Dynamic events in the flexible job-shop scheduling problem: Rescheduling with a hybrid metaheuristic algorithm”. In: *Algorithms* 17.4 (2024), p. 142. DOI: 10.3390/a17040142. URL: https://www.mdpi.com/1999-4893/17/4/142.

[25] Richard S Sutton and Andrew G Barto. “Reinforcement learning: An introduction”. In: 6 (2021).

[26] William Fedus et al. “Revisiting fundamentals of experience replay”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 3061–3071.

[27] Volodymyr Mnih et al. *Playing Atari with deep reinforcement learning*. Dec. 2013. DOI: 10.48550/arXiv.1312.5602. URL: http://arxiv.org/abs/1312.5602 (visited on 04/23/2025).

[28] Stefano V Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-agent reinforcement learning: Foundations and modern approaches*. MIT Press, 2024.

[29] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676 (2017), pp. 354–359.

[30] Volodymyr Mnih et al. “Asynchronous methods for deep reinforcement learning”. In: *International Conference on Machine Learning*. PmLR. 2016, pp. 1928–1937.

[31] John Schulman et al. *Proximal policy optimization algorithms*. Aug. 2017. DOI: 10.48550/arXiv.1707.06347. URL: http://arxiv.org/abs/1707.06347 (visited on 07/20/2025).

[32] Tong Zhou et al. “Reinforcement learning for online optimization of job-shop scheduling in a smart manufacturing factory”. In: *Advances in Mechanical Engineering* 14.3 (Mar. 2022), p. 16878132221086120. DOI: 10.1177/16878132221086120. URL: https://journals.sagepub.com/doi/10.1177/16878132221086120 (visited on 03/26/2025).

[33] Cong Zhang et al. *Learning to dispatch for job shop scheduling via deep reinforcement learning*. 2020. DOI: 10.48550/ARXIV.2010.12367. URL: https://arxiv.org/abs/2010.12367 (visited on 03/26/2025).

[34] Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. “An end-to-end reinforcement learning approach for job-shop scheduling problems based on constraint programming”. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 33 (July 2023), pp. 614–622. DOI: 10.1609/icaps.v33i1.27243. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/27243 (visited on 07/20/2025).

[35] Chinyere Ngwu, Ying Liu, and Rui Wu. “Reinforcement learning in dynamic job shop scheduling: A comprehensive review of AI-driven approaches in modern manufacturing”. In: *Journal of Intelligent Manufacturing* (Mar. 2025). DOI: 10.1007/s10845-025-02585-6. URL: https://link.springer.com/10.1007/s10845-025-02585-6 (visited on 03/26/2025).

[36] Petar Veličković. “Everything is connected: Graph neural networks”. In: *Current Opinion in Structural Biology* 79 (2023), p. 102538.

[37] Zhenyu Liu et al. “Dynamic job-shop scheduling using graph reinforcement learning with auxiliary strategy”. In: *Journal of Manufacturing Systems* 73 (2024), pp. 1–18.

[38] Igor G. Smit et al. “Graph neural networks for job shop scheduling problems: A survey”. In: *Computers & Operations Research* 176 (Apr. 2025), p. 106914. DOI: 10.1016/j.cor.2024.106914. URL: https://www.sciencedirect.com/science/article/pii/S0305054824003861 (visited on 07/20/2025).

[39] Xiangyu Zhao et al. “Embedding in recommender systems: A survey”. In: *arXiv preprint arXiv:2310.18608* (2023).

[40] Fouad Bahrpeyma and Dirk Reichelt. “A review of the applications of multi-agent reinforcement learning in smart factories”. In: *Frontiers in Robotics and AI* 9 (Dec. 2022), p. 1027340. DOI: 10.3389/frobt.2022.1027340. URL: https://www.frontiersin.org/articles/10.3389/frobt.2022.1027340/full (visited on 03/26/2025).

[41] Libing Wang et al. “Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning”. In: *Computer Networks* 190 (May 2021), p. 107969. DOI: 10.1016/j.comnet.2021.107969. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128621001031 (visited on 03/26/2025).

[42] Xinquan Wu et al. “A deep reinforcement learning model for dynamic job-shop scheduling problem with uncertain processing time”. In: *Engineering Applications of Artificial Intelligence* 131 (May 2024), p. 107790. DOI: 10.1016/j.engappai.2023.107790. URL: https://linkinghub.elsevier.com/retrieve/pii/S0952197623019747 (visited on 03/26/2025).

[43] Lingling Lv et al. “A multi-agent reinforcement learning based scheduling strategy for flexible job shops under machine breakdowns”. In: *Robotics and Computer-Integrated Manufacturing* 93 (2025), p. 102923.

[44] Eric Taillard. “Benchmarks for basic scheduling problems”. In: *European Journal of Operational Research* 64.2 (1993), pp. 278–285.

[45] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. “Benchmarks for shop scheduling problems”. In: *European Journal of Operational Research* 109.1 (1998), pp. 137–141.

[46] Kuo-Hao Ho et al. “Residual scheduling: A new reinforcement learning approach to solving job shop scheduling problem”. In: *IEEE Access* 12 (2024), pp. 14703–14718.

[47] Chien-Liang Liu and Tzu-Hsuan Huang. “Dynamic job-shop scheduling problems using graph neural network and deep reinforcement learning”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 53.11 (Nov. 2023), pp. 6836–6848. DOI: 10.1109/TSMC.2023.3287655. URL: https://ieeexplore.ieee.org/document/10176351/ (visited on 03/26/2025).

[48] Yufeng Li et al. “Flexible job-shop scheduling via graph neural network and deep reinforcement learning”. In: *IEEE Transactions on Industrial Informatics* 18.9 (2022). Accessed: Apr. 02, 2025, pp. 6477–6487. DOI: 10.1109/TII.2022.3163266. URL: https://ieeexplore.ieee.org/document/9826438.

[49] OASIS Standard. “MQTT version 3.1. 1”. In: *URL http://docs. oasis-open. org/mqtt/mqtt/v3* 1 (2014), p. 29.

[50] Lukas Biewald. *Experiment tracking with Weights and Biases*. https://wandb.ai/site. Software available from https://wandb.ai. 2020.