FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Graph neural networks and deep reinforcement learning in job-shop scheduling
Student:	Bc. Yury Hayeu
Supervisor:	prof. Ing. RNDr. Martin Holeňa, CSc.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2024/2025

Instructions

One of the most common problems in industries such as manufacturing and transportation [1] is the assignment of large amounts of operations to available resources, i.e. processing units, in such a way that the order of processed operations and the specification of processing units are respected. The solution to the problem is the schedule that is optimal to business-aligned criteria, e.g. the time of completion, manufacturing cost, utilization rate and others. The problem is known as a job-shop scheduling problem (JSSP). It is commonly solved using heuristic methods such as priority dispatch rules, evolutionary algorithms, ant colony optimisation and others. However, these methods use the assumptions of a static environment and invariable problem specification that compromises performance in practice. Recent studies [2-7] claim that methods based on Reinforcement Learning (RL) perform well in dynamic environments with random job arrival. However, the application of RL-based methods to JSSP is complicated due to flexibility in state and action spaces. One of the prominent research directions is to represent the state of the shop floor using a graph and later process it with graph neural networks (GNN) [4, 6] to obtain a fixed-length representation. There is a lack of a unified benchmarks to evaluate the performance of methods under dynamic conditions. In most cases, the authors either implement their dynamic simulator or evaluate it on static instances.

The student has to accomplish the following tasks:
1. Get familiar with methods of solving JSSP employing RL and Graph Neural Networks.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 16 February 2024 in Prague.

2. Implement a simulator to evaluate the performance of JSSP under random job arrival. The simulator must support basic JSSP and some possible extensions, e.g. flexible JSSP, JSSP with setup times, JSSP with breakdowns and others.
3. Select and implement at least 5 methods you learned in 1.
4. Propose a novel method. Focus on
- the improvement of current methods by incorporating ideas from research in RL or GNN;
- incorporating information about system evolution using recurrent neural networks or dynamic graph networks.
5. Propose an evaluation pipeline and use it to perform a comparative study of the implemented methods.

Resources:
1. The flexible job shop scheduling problem: A review: https://www.sciencedirect.com/science/article/pii/S037722172300382X
2. A deep multi-agent reinforcement learning approach to solve dynamic job shop scheduling problem: https://dl.acm.org/doi/10.1016/j.cor.2023.106294
3. An End-to-end Hierarchical Reinforcement Learning Framework for Large-scale Dynamic Flexible Job-shop Scheduling Problem: https://ieeexplore.ieee.org/document/9892005
4. Combining Reinforcement Learning Algorithms with Graph Neural Networks to Solve Dynamic Job Shop Scheduling Problems": https://www.mdpi.com/2227-9717/11/5/1571
5. Deep reinforcement learning for dynamic flexible job shop scheduling problem considering variable processing times: https://www.sciencedirect.com/science/article/abs/pii/S0278612523001917
6. Dynamic job-shop scheduling using graph reinforcement learning with auxiliary strategy: https://www.sciencedirect.com/science/article/abs/pii/S0278612524000025
7. Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning: https://www.sciencedirect.com/science/article/abs/pii/S1568494620301484

Master's thesis

GRAPH NEURAL NETWORKS AND DEEP REINFORCEMENT LEARNING IN JOB-SHOP SCHEDULING

Bc. Yury Hayeu

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: prof. Ing. RNDr. Martin Holeňa, CSc.
May 9, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Hayeu Yury. *Graph neural networks and deep reinforcement learning in job-shop scheduling.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of Algorithms

Throughout the writing of the thesis, I have received a great deal of support and assistance.

I would first like to thank my supervisor, prof. Ing. RNDr. Martin Holeňa, CSc. for his guidance.

I would also like to thank my family for their support and encouragement throughout my studies. I also gratefully acknowledge the assistance of my brother, who helped me with the continuous proofreading of the thesis.

Finally, I wish to thank the CTU FIT staff for being so helpful and supportive during the two years of my study.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 9, 2024

Abstract

Dynamic Job Shop Scheduling Problem is an NP-hard optimization problem with a wide range of real-world applications. One of the ways to approach the problem is to use priority dispatch rules, which are simple heuristic functions to compute job priority. Priority dispatch rules are known to behave well in some but not all applications and their performance deteriorates over time in the dynamic environment. Recent methods mitigate the issue by learning how to adaptively select the optimal rule. Some of them are based on Reinforcement Learning and Graph Neural Networks. The thesis studies the performance of these methods in a dynamic environment with machine breakdowns with the objective of tardiness minimization.

Keywords Dynamic Job Shop Scheduling Problem, Deep Reinforcement Learning, Graph Neural Networks, Machine Breakdown, Tardiness Minimization

Abstrakt

Problém dynamického rozvrhování úloh je NP-těžký optimalizační problém s širokou škalou reálných aplikací. Jedním ze způsobů, jak přistoupit k problému dynamického rozvrhování úloh, je použití pravidel prioritního rozvrhování, což jsou jednoduché heuristické funkce pro výpočet priority úlohy. Je známo, že pravidla prioritního rozvrhování se v některých, ale ne ve všech aplikacích chovají dobře a jejich výkon se v dynamickém prostředí s časem zhoršuje. Nejnovejší metody přístupují k problému tak, že se učí adaptivně vybírat nejlepší pravidlo. Některé z nich jsou založené na posilovaném učení a grafových neuronových sítích. Tato práce studuje výkon těchto metod v dynamickém prostředí s poruchami strojů s cílem minimalizovat zpoždění provedení práce.

Klíčová slova Dynamický Rozvrhovací Problém, Hluboké Posilované Učení, Grafové Neuronové Sítě, Porucha Strojů, Minimalizace Zpoždění

List of abbreviations

ACO	Ant Colony Optimization
DQN	Deep Q-Network
DDQN	Double Deep Q-Network
DJSSP	Dynamic Job Shop Scheduling Problem
DRL	Deep Reinforcement Learning
FJSSP	Flexible Job Shop Scheduling Problem
GA	Genetic Algorithm
GAT	Graph Attention Network
GATv2	Graph Attention Network v2
GraphSAGE	Graph Sample and Aggregate
GCN	Graph Convolutional Network
GIN	Graph Isomorphism Network
GP	Genetic Programming
GNN	Graph Neural Network
JK	Jumping Knowledge
LWKR	Least Work Remaining Priority Dispatch Rule
MDP	Markov Decision Process
MLP	Multi-Layer Perceptron
MS	Minimum Slack Priority Dispatch Rule
MPP	Message Passing Paradigm
NPT	Natural Policy Gradient
PDR	Priority Dispatch Rule
PSO	Particle Swarm Optimization
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SPT	Shortest Processing Time Priority Dispatch Rule
SVM	Support Vector Machine
TRPO	Trust Region Policy Optimization
WINQ	Work In Next Queue Priority Dispatch Rule
Deep MARL	Deep Multi-Agent Reinforcement Learning
Deep MARL-AS	Deep Multi-Agent Reinforcement Learning Abstract State
Deep MARL-MR	Deep Multi-Agent Reinforcement Learning Minimum Repetition

Introduction

The problem of the assignment of a large number of operations to available processing units is one of the most common problems in industries such as manufacturing and transportation. The problem is also known as the Job Shop Scheduling Problem (JSSP). The JSSP by definition is a static problem, i.e. no dynamic factors (e.g. new job arrivals, machine breakdowns, etc.) are considered, which limits the applicability of the methods to real-world scenarios. The extension of the JSSP, which considers various dynamic factors, is known as the Dynamic Job Shop Scheduling Problem (DJSSP). One of the ways to approach the DJSSP is to use a simple heuristic function to compute the priority of the jobs, known as priority dispatch rules (PDRs). PDRs are simple to implement, however, their performance is known to deteriorate in dynamic environments. Therefore, modern methods focus on the automatic selection of the optimal rule based on the current state of the shop floor. One group of these methods learns to select the optimal PDR using Deep Reinforcement Learning (DRL) methods. Some methods combine Graph Neural Networks (GNNs) with DRL to represent the state of the system. The thesis focuses on the study of methods employing DRL and GNNs for the DJSSP to minimize a total production delay (total cumulative tardiness).

The theoretical part starts with the definition of the Job Shop Scheduling Problem and its most common variants. The thesis primarily focuses on the DJSSP, therefore, after the introduction, the narrative proceeds with an overview of methods applied to solve the problem. The description is concluded with a discussion of methods to benchmark the DJSSP solver. The theoretical part then introduces the basic concepts of RL. After the basics are introduced, the description continues with a detailed explanation of DRL methods. Following DRL methods, GNNs are discussed in detail. Finally, the theoretical part ends with a description of methods applied to solve the DJSSP problem using RL and GNNs and a discussion of the evaluation procedure proposed in the scope of the thesis.

The practical part of the thesis starts with a series of experiments to gain insights into the performance of the studied methods. At the end of the prac-

tical part, the insights are combined to perform the final evaluation of the studied methods.

Goals

The main goal of the thesis is to evaluate the performance of methods employing Deep Reinforcement Learning and Graph Neural Networks to solve the Dynamic Job Shop Scheduling Problem with the objective of the minimization of total cumulative tardiness. After the initial study of the literature, we found it challenging to perform an exact comparison of different approaches due to their focus on specific applications and, in some cases, limited availability of the source code. Therefore, the goal of the thesis is pointed out as follows:

- RL methods learn to select the action that achieves the best overall outcome by observing the state of the system. Some of the studied methods use a set of tailored indicators to describe the state of the system, which rules out the possibility of reusing publicly available simulators for the JSSP. Hence, the first subgoal is to implement a simulator for the DJSSP that can be used to evaluate the performance of the studied methods.
- Most of the studied methods don't focus on the minimization of the tardiness of job production. Hence, the second goal is to identify the most important components of the studied methods and propose the set of approaches that must be evaluated in the scope of the thesis.
- Finally, the proposed set of approaches must be evaluated on the implemented simulator. It is important to note, that RL methods are sensitive to the initializations and hyperparameter configurations, therefore, the evaluation must be designed to ensure the reliability of the results.

Chapter 1

Job Shop Scheduling Problem

Job Shop Scheduling Problem (JSSP) is one of the most frequently encountered combinatorial optimization problems in such industries as manufacturing, transportation, etc. [1, 2] The wide range of applications makes it one of the most studied problems in the field of operational research. Research in JSSP extends beyond the standard definition, encompassing modifications, and extensions adapted to specific application needs. This and long research history, dating back to the 1960s, has led to the development of a wide range of methods to solve the problem. The goal of the chapter is to introduce the standard definition of the problem, define a set of modifications and extensions related to the thesis, and provide a brief overview of methods used to solve the problem.

1.1 Definition

JSSP is a classic optimization problem [3]. Let $\mathcal{J}$ be the set of jobs and $\mathcal{M}$ be the set of machines on the shop floor. Each job $j \in \mathcal{J}$ has a sequence of associated operations $o_{ij} \in \mathcal{O}_j$. Each operation o_{ij} can be processed on a specific machine $m_k \in \mathcal{M}$ with processing time p_{o_{ij},m_k}. The goal is to find an assignment of starting times for each operation that minimizes the given objective function. Furthermore, the assignment must satisfy the following set of constraints [4].

- C1: All jobs are simultaneously available for processing (at time 0).
- C2: Machines can process only one operation at a time.
- C3: Machines setup time is independent of job sequence and incorporated in operation processing time.
- C4: Machines are continuously available (no breakdowns occur).
- C5: Machines never idle while work is waiting.
- C6: Once an operation starts, it proceeds without interruptions.

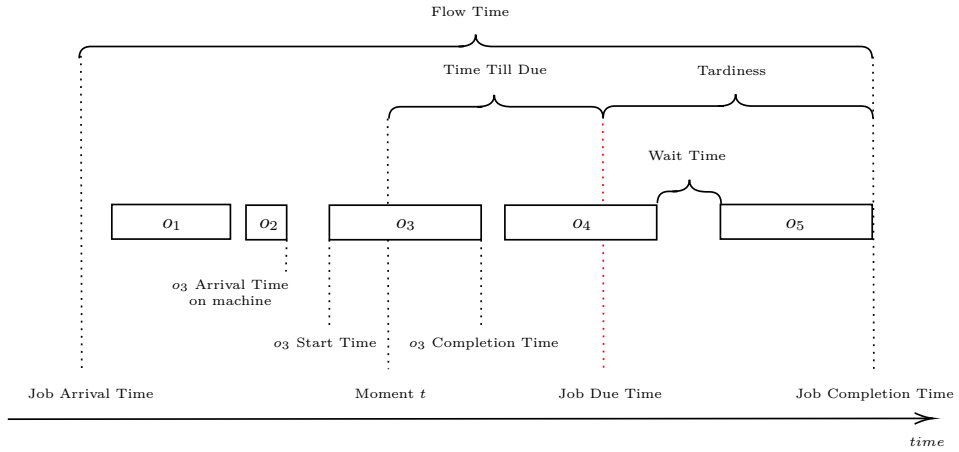


Figure 1.1 Flow of a job consisting of 5 operations.

The JSSP problem is known to be NP-hard for $|\mathcal{M}| \geq 3$ [5] meaning that the optimal solution is usually intractable. To obtain sub-optimal solutions, a wide range of heuristics and metaheuristics were developed. These methods usually attempt to solve the problem locally relying on domain-specific information. To describe these methods a set of job-, machine- and shopfloor-specific definitions is needed.

The flow of a job consisting of 5 operations is presented in Figure 1.2. When a job arrives at the shop floor it is immediately dispatched to the machine. The machine receives the job and starts processing if it is available. Otherwise, the job is stored in the machine queue until the machine decides to process it. After the processing of the operation, the job is forwarded to the next machine. The process is repeated until all operations are completed. One of the natural methods to incorporate job priorities is to assign due time to each job. The job not completed by the due time is called tardy. The duration between the due time and the completion time for a tardy job is called tardiness. If the job is completed before the due time, the job is early and the duration is denoted as earliness. The time the job spends on the shop floor is called flow time. The notation of the variables and further definitions are presented in Table 1.2.

1.2 Dynamic Job Shop Scheduling

Following the (C1) constraint, all jobs must be simultaneously available for processing at the beginning of the simulation. While this assumption simplifies the research of the problem, it is a significant limitation for applications in real-world environments that are continuously affected by dynamic factors. By relaxing the aforementioned constraint, the set of $\mathcal{J}$ jobs presented on the shopfloor changes over time. Furthermore, researchers relax (C4), (C5), and (C6) constraints by considering machine breakdowns, idle time, and job

Table 1.2 List of terms and variables used for the description of JSSP.

J_i	i-th job
$\mathcal{J}$	Set of all jobs
$\mathcal{J}^k$	Set of all jobs in the queue of k-th machine
m_i	i-th machine
$\mathcal{M}$	Set of all machines
O_i	Number of operations of i-th job
$o_{i,j}$	i-th operation of j job
$p_{i,k}$	Processing time of J_i on machine m_k
p_i^j	Processing time of j-th operation of J_i
D_i	Due time of J_i
NOW	Current time in the system
TTD_i	Time till due, $TTD_i = D_i - NOW$
WR_i	Remaining time of the job, $WR_i = \sum_{j=k}^{O_i} p_i^j$ where k is the index of J_i's next operation
S_i	Slack time of J_i, $S_i = TTD_i - WR_i$
$cr_{i,j}$	Completion rate of j-th job at i-th operation, i.e. $cr_i = i/O_i$
$a_{i,k}$	Arrival time of J_i on machine k
$r_{i,k}$	Release time of J_i on machine k
R_i	Release time of J_i
C_i	Completion time of J_i
L_i	Lateness of J_i, $L_i = C_i - D_i$
T_i	Tardiness of J_i, $T_i = \max(C_i - D_i, 0)$
T_{sum}	Cumulative tardiness, $T_{sum} = \sum_i^{\mathcal{J}} T_i$
E_i	Earliness of J_i, $E_I = \min(D_i - C_i, 0)$
E_{sum}	Cumulative earliness, $E_{sum} = \sum_i^{\mathcal{J}} E_i$
F_i	Flow time of J_i, $F_i = C_i - R_i$

cancellations [6]. The JSSP problem with dynamic events is then called the Dynamic Job Shop Scheduling Problem (DJSSP).

DJSSP instance can be decomposed into a set of static instances which also means that the problem remains NP-hard [7]. In comparison to the static problem, the dynamic variant requires the method to continuously reconsider the schedule and adapt to the changing environment.

1.3 Criteria

One of the essential components of the optimization task is the criterion function used to evaluate the quality of the solution. For JSSP, optimization criteria are commonly represented by functions of completion times.

$$\mathcal{O} = f(C_1, C_2, \ldots, C_n), \qquad i = 1, 2, \ldots, |\mathcal{J}| \tag{1.1}$$

Furthermore, these measures belong to an important class of criteria called regular criteria.

▶ **Definition 1.1** (Regular Criterion)**.** *Let $\mathcal{O}$ be a function of completion times. $\mathcal{O}$ is regular if the optimization task minimizes $\mathcal{O}$ and $\mathcal{O}$ can increase only if at least one of the completion times in the schedule increases.*

By considering regular criteria we restrict attention to the set of schedules called a *dominant* set. To verify that a set of schedules is dominant, it is required to show that for any possible schedule, there exists a better schedule from the dominant set. For example, by relaxing the (C6) constraint the pre-emption of operations can be allowed, but it would never lead to a better solution, because the set of solutions with the restriction is known to be dominant [4].

One of the most common and most studied regular criteria is the maximum completion time of all jobs in the system or makespan which is defined as follows.

$$C_{max} = \max_{i \in \mathcal{J}} C_i$$

Under the assumption of static JSSP, the makespan represents an intuitive measure of the schedule quality. However, in dynamic environments, the makespan is a misleading measure in some scenarios. For example, one such scenario is the late arrival of the long-processing job. The makespan is equal to the completion time of the late-arrived job which doesn't reflect the quality of the schedule [8]. Hence, the makespan criterion should be cautiously used in dynamic environments.

The other regular criterion more suitable for dynamic environments is total weighted flow time (TWF) which is defined as follows.

$$TWF = \sum_{i \in \mathcal{J}} w_i F_i$$

The measure is suitable when the quality of the solution is directly related to the speed of job processing by the system. It is usually achieved by neglecting the priority of the job. For instance, let's consider JSSP with due dates which contains a large number of small jobs and one already tardy job. The model minimizing the TWF criterion prioritizes the job with the smallest flow time through the system over the tardy one.

If the extreme delays in job completion must be avoided, the maximum or cumulative tardiness (T_{sum}) criteria can be used.

$$T_{max} = \max_{i \in \mathcal{J}} T_i$$

Finally, if the penalty doesn't depend on the delay, the weighted number of tardy jobs (TWT) can be used. The criterion is known to be equivalent to the maximization of system throughput [9].

$$TWT = \sum_{i \in \mathcal{J}} T_i$$

In general, regular criteria that focus on the completion times fall short of capturing how the individual operations are performed. This led to the development of non-regular criteria which took a step away from the idea that criterion must be represented as a function of job completion times. The group of just-in-time criteria is one of the representatives of non-regular criteria. One of the just-in-time criteria is an Earliness-Tardiness criterion which is defined as a linear combination of total weighted earliness and tardiness [6].

$$E/T = \sum_{i \in \mathcal{J}} w_i T_i + \sum_{i \in \mathcal{J}} w_i E_i$$

Overall, optimization criteria are the essential part of the optimization task which defines the expected behavior of the solution. Research in the field of JSSP mainly focuses on the makespan criterion while other criteria are significantly less studied. However, the makespan criterion is not suitable for dynamic environments which complicates the application of static methods in real-world scenarios.

1.4 Flexible Job Shop Scheduling

Flexible Job Shop Scheduling Problem (FJSSP) is a generalization of the JSSP problem where each operation can be processed by any machine from a fixed set of machines arranged in the work center. Similarly to JSSP, the job must visit each work center in a specific order to be completed, but it must be processed by only one machine in each work center. The main prerequisite of the FJSSP is the heterogeneity of processing times across machines. Otherwise, the work center can be represented as a single machine with an increased capacity of memory which leads to the standard JSSP problem. The FJSSP problem also belongs to the set of NP-hard problems. However, the problem is significantly more complicated due to the additional degree of freedom in the selection of the best machine suitable for processing the operation at the moment.

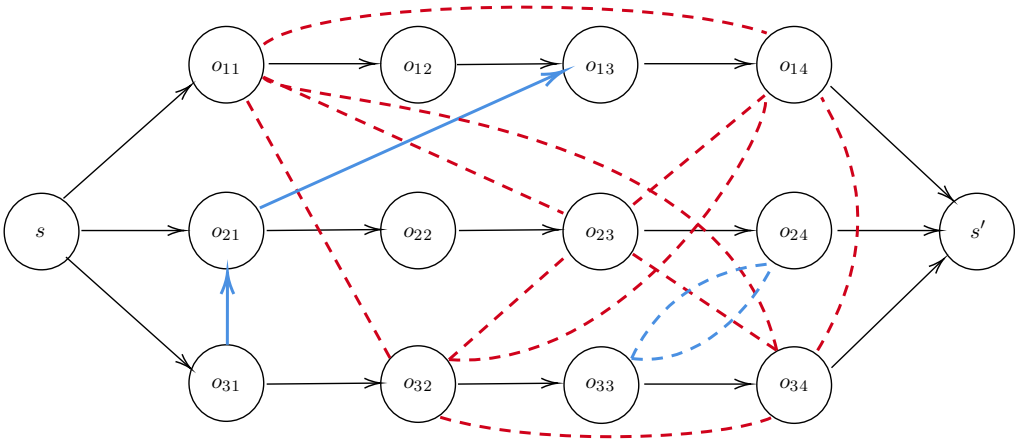


Figure 1.2 The disjunctive graph of JSSP instance of 3 jobs on the shop floor with 3 machines. It is assumed that the job has several operations processed on the same machine. Dashed edges represent the disjunctive graph for a single machine, while colored directed edges represent the partial solution of the JSSP instance.

1.5 Graph Representation

JSSP is often modeled as a disjunctive graph $G = (V, E)$.

▶ **Definition 1.2** (Disjunctive Graph)**.** *The disjunctive graph is a directed graph* $G = (V, C \cup D)$ *where*

- V *is a set of vertices consisting of nodes for job operations and source* s *and sink* s' *node* ($V = \{\forall J_i \in \mathcal{J}, \forall o_{ij} \in J_i : o_{ij}\} \cup \{s, s'\}$).
- C *is the set of directed edges (conjunctions) representing the precedence constraints between operations of a single job.*
- D *is the set of undirected edges (disjunctions) between operations executed on the same machine.*

The solution of the JSSP is equivalent to selecting the direction of all edges in the disjunctive graph without introducing any cycles. Note, that the disjunctive graph is formed by the union of $|\mathcal{M}|$ complete graphs, where each graph has at most $|\mathcal{J}|$ nodes. The disjunctive graph representation can be applied to the FJSSP problem. Each job operation is represented with a set of nodes, one for each machine in the work center. The solution of the FJSSP is equivalent to the construction of $|\mathcal{M}|$ paths from source to sink node of length at most $|\mathcal{J}|$ without introducing any cycles. The constructed set of paths must cover all operations for all jobs.

1.6 Dynamic Job Shop Scheduling Methods

Methods for solving dynamic JSSP are usually separated into the following categories [10]:

- **Reactive Methods:** These methods perform scheduling based on real-time information of the system. Hence they are suitable for both static and dynamic environments. The main representative category of reactive methods is priority dispatch rules (PDRs). While most of the PDRs are simple and easy to implement, they are unable to provide strong performance on all objectives in all scenarios [11, 12].
- **Robust-proactive Methods:** These methods usually employ metaheuristics techniques used for solving the static JSSP instances. The solution development starts with a population of candidate schedules which are evaluated in dynamic environments under various scenarios. Candidates with better performance are replicated and mutated in iterative evolution. The goal is to find a solution robust to various disruptions on the shop floor. The obtained solution is usually more complete and less myopic but to process of finding the solution has a high computational cost.
- **Proactive Reactive:** These methods construct a schedule by considering a static version of the JSSP problem. When a disruption occurs, the method either corrects the schedule to obtain the sub-optimal solution or reconstructs it based on the new system state. The main challenge of these methods is to assess the severity of the disruption whether the schedule should be reconstructed or the corrected version is satisfactory.

1.7 Reactive Methods

One of the main representatives of reactive methods is priority dispatch rules (PDRs). When a machine completes an operation of the job, the priority dispatch rule selects the next operation to be processed. The selection is done by calculating the priority for each job in the machine queue and selecting one with the best value. Alternatively, the rule can decide to idle, but under the scope of regular schedules, the decision is obsolete. There are several ways to classify PDRs.

First, the rules are classified depending on the locality of information they use to make a decision. Local rules employ only the information about jobs stored in the machine's memory. For instance, rules, which select the job with the smallest available slack time or the job with the shortest processing time are local. In contrast, global rules consider the state of other machines on the shop floor. For example, the Work In Next Queue (WINQ) rule selects the job that after processing is dispatched to the machine with the smallest workload. From the performance perspective, global rules should perform better than local

ones but there is no evidence to that claim. From the number of considered parameters perspective, global rules have more information about the shop floor state which usually leads to higher complexity from the implementation perspective.

The second method of PDRs classification is based on the dynamics of the input information. The rule is static if the information doesn't change over time and otherwise dynamic. For instance, the Earliest Due Date belongs to the class of static rules. Certain rules are static to operations of the job, e.g. the Shortest Processing Time (SPT) rule or the Least Work Remaining Rule (LWKR). The dynamic rules usually incorporate information related to the due time of the job (slack time, tardiness, etc.)[13].

Finally, rules can be classified based on the information they are using to make a decision. These rules are usually designed to target specific optimization criteria. For instance, the rule that doesn't consider the due time of the job performs poorly for the tardiness criterion.

Research of PDRs is directed towards the development of new methods based on PDRs that can adapt to various scenarios. In dynamic environments, even the optimal selection of PDRs deteriorates over time. One of the ways to mitigate the issue is to perform an adaptive selection of PDRs. The idea can be further expanded by selecting a set of rules for individual machines on the shop floor in a multi-agent fashion. Adaptive simulation-based optimization [14] method employs the simulation-assisted genetic algorithm for selecting the optimal rule for each machine. Most recent methods employ Deep Reinforcement Learning (DRL) for adaptive selection of PDRs. A detailed overview of these methods is covered in Chapter 4.

The other direction of research is the manual or automatic design of PDRs. These methods are usually built on the premise of simplicity for rule combinations. For instance, the SPT rule can be adapted to due dates by adding slack time for each job in the queue. The process of manual rule design is tedious and requires a deep understanding of relations between system parameters. Modern methods attempt to design new PDRs automatically. The majority of such methods are based on Genetic Programming (GP). GP is an evolutionary algorithm to automatically find computer programs to solve specific tasks. For JSSP the individual of the population is represented with a tree structure where inner nodes are operands and leaves are parameters of the job. In comparison to manually designed rules, rules obtained by GP possess a high level of adaptability to various scenarios and consider a wider range of system parameters. The other advantage is the high interpretability of rules obtained by GP. GP-generated rules can be used as basic building blocks of adaptive methods discussed in the previous paragraph for further possible performance gains [15]. Further details on the methods of automatic rules design can be found in [16, 17].

For the scope of the thesis a wide range of PDRs initially studied in [8] and further expanded by us is presented in Table 1.3.

Table 1.3 Overview of PDRs evaluated in the scope of the thesis. The rules are further classified by scope information (L: ✓- local vs. ✗- global), by structure (S: ✓- simple vs. ✗- composite), by involvement of processing time (P: ✓- with vs. ✗- without), and by due date information (D: ✓- with vs. ✗- without).

Rule	L	S	P	D
Apparent Tardiness Cost (ATC) [18]	✓	✓	✓	✓
Average Processing Time per Operation (AVPRO)	✓	✓	✓	✗
Cost Over Time (COVERT)	✓	✓	✓	✓
Critical Ratio (CR)	✓	✓	✓	✓
Earliest Due Date (EDD)	✓	✓	✗	✓
First In First Out (FIFO)	✓	✓	✗	✗
Generatic Programming 1 (GP1) [19]	✗	✗	✓	✗
Generatic Programming 2 (GP2) [20]	✗	✗	✓	✗
Last In First Out (LIFO)	✓	✓	✗	✗
Longest Processing Time (LPT)	✓	✓	✓	✗
Least Remaining Operations (LRO)	✓	✓	✗	✗
Least Work Remaining (LWKR)	✓	✓	✓	✗
Longest Waiting Time (LWT)	✓	✓	✓	✗
Modified Due Date (MDD)	✓	✓	✓	✓
Modified Operational Due Date (MOD)	✓	✓	✓	✓
Montagne Heuristics (MON)	✓	✓	✓	✓
Most Remaining Operations (MRO)	✓	✓	✗	✗
Minimum Slack (MS)	✓	✓	✓	✓
Most Work Remaining (MWKR)	✓	✓	✓	✗
Next Processing Time (NPT)	✓	✓	✓	✗
Random	✗	✓	✗	✗
Slack per Remaining Work (SPW)	✓	✓	✓	✓
Shortest Processing Time (SPT)	✓	✓	✓	✗
Shortest Waiting Time (SWT)	✓	✓	✓	✗
Work In Next Queue (WINQ)	✗	✓	✓	✗
CR + SPT	✓	✗	✓	✓
2PT+ LWKR	✓	✗	✓	✓
2PT + LWKR + Slack	✓	✗	✓	✗
2PT + WINQ + NPT	✗	✗	✓	✗
LWKR + MOD	✓	✗	✓	✓
LWKR + SPT	✓	✗	✓	✗
SPW + SPT	✓	✗	✓	✓
PT + WINQ	✗	✗	✓	✗
PT + WINQ + Slack	✗	✗	✓	✗

As we discussed in previous sections, the solution of the FJSSP involves an additional complexity. When the work center receives a job or a batch of jobs, it must select the best machine suitable for job processing. Literature distinguishes two types of methods applied to solve the FJSSP problem: integrated [21] and hierarchical [22]. At first, integrated methods try to avoid complex decisions on the work center level by selecting the machine with the Earliest Available (EA) rule. By doing so, these methods assume that the processing power of each machine is the same which isn't usually true. Hierarchical methods use an additional routing rule to select the best machine suited for the processing of the job. Research methods and directions of routing rules are similar to methods in PDR research discussed earlier. It is worth mentioning that the simultaneous optimization of routing and dispatching rules possesses a high level of complexity. Hence, earlier studies focus on the integrated methods [12]. However, modern methods usually focus on the hierarchical approach. For instance, authors in [23] propose a cooperative coevolution genetic programming framework, which evolves routing and dispatching rules in two sub-populations simultaneously.

As we discussed earlier, the goal of robust proactive methods is to find a solution that is robust to a wide range of disruptions on the shop floor. To be more specific robustness is divided into two groups [24]: *solution robustness* and *quality robustness*. *Quality robustness* refers to the sensitivity of the solution performance in terms of the objective function, while *solution robustness* is used to describe the insensitivity of the activity start times to variations in input data. Robustness is also important for proactive reactive methods because robust solutions can be usually corrected when a disruption occurs [25].

To create a robust schedule two factors must be considered. At first, the quality of the solution is directly related to the set of disruptions the method is created for. While it is possible to cover a wide range of them, the method robust to all disruptions appearing in the real-world scenario is intractable. Secondly, the measure of robustness must be defined. Literature [26, 27] distinguishes two types of robustness measures:

- **Scenario-based measures:** The schedule is evaluated under various scenarios for which a deviation in the performance is measured. A small deviation indicates a highly robust solution. These measures are known to be effective albeit with high computational costs [28].
- **Surrogate measures:** These methods measure a set of various indicators for the schedule. Then by considering the measurements, it is possible to estimate the robustness of the solution. Commonly indicators are the functions of slack time [27].

Metaheuristics are the most common methods employed to find a robust solution for JSSP instances. They represent a wide set of problem-agnostic

frameworks that are applied to develop heuristic optimization methods. All of these methods consist of two main components: diversification and intensification. During the diversification phase, the method explores the search space globally to find the region of good solutions, while during the intensification phase, the method focuses on searching for the best solution in the obtained region. For a good rate of convergence, the balance between both phases must be maintained. Depending on the number of solutions metaheuristics operate on they can be divided into local and global methods. Local methods, for instance, the Simulated Annealing (SA) or the Tabu Search (TS), develop a single solution by searching a region of states *close* in some matter to the current solution. On the contrary, global methods, for instance, Genetic Algorithm (GA), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO) deal with a set of solutions, that allow the exploration of distinct search space regions simultaneously [29].

In the context of JSSP, the most common metaheuristics applied is GA. The method starts with the initial population of schedules usually represented in the form of ordered assignment of operations that must processed by the machine. The quality of each individual from the population is measured by the objective or fitness function. Then the search is performed by exchanging parts of the solutions between *good* (in terms of fitness function) individuals (crossover) or random adjustments of the individual (mutation). From the perspective of finding robust solutions, the fitness function is usually a combination of the criterion and surrogate robustness measure. Hence, the extensions of the GA for multi-objective optimization are usually employed. For instance, authors of the [28] develop two surrogate measures and study their effectiveness for FJSSP with machine breakdowns using non-dominated sorting genetic algorithm II (NSGA-II) [30]. Other applications include a non-dominated ranking genetic algorithm [31] for FJSSP [32], a two-stage-genetic algorithm for FJSSP with breakdowns [33], etc.

ACO imitates the behavior of ants to find the shortest path between the source and the origin. The first application of ACO to solve the JSSP problem was developed in the year 1994 [34], by visiting nodes of the disjunctive graph the ant constructs the solution. The behavior of ants is later tuned using a pheromone-based mechanism. The method was later generalized to FJSSP [35]. In the context of robust proactive methods, authors of [36] propose a set of improvements for the ACO method to solve robust FJSSP problem [36]. The other application can be seen in [37], in which authors enhance the standard ACO method with a local search method to obtain a robust schedule for hot-rolling production.

1.8 Proactive Reactive Methods

Proactive reactive methods construct the solution relying on the current state of the shop floor without considering any future events, i.e. the problem re-

sembles static scheduling. When an event is received, the method corrects the solution in response. The type of correction is directly related to the type of event. For instance, the arrival of a new job with a late due time can be resolved by delaying the processing of the arrived job until the next construction of the schedule, while the machine breakdown requires immediate schedule reconstruction. In practice, most events arrive on the shop floor with high frequency and have a local impact (e.g. job creation, job modification, job cancellation, etc.) Under such conditions, complete reconstruction is at most as difficult as the complexity of the method employed for initial schedule creation. Commonly, reconstruction possesses high computational cost, hence, repair methods are widely applied to obtain sub-optimal yet feasible solutions without significant delays. Simple heuristics are employed in practice because they produce stable schedules. The examples are right-shift [38] rescheduling, which postpones unfinished jobs for the disruption time, and left-shift rescheduling [39], which shifts the start times of unfinished jobs to the moment of the disruption. Note, that both of these methods preserve the schedule before the disruption. Due to the locality of the impact, some repair methods aim to perform rescheduling only for disrupted parts of the shop floor. For instance, the matchup [40] method performs rescheduling to restore the schedule before the disruption at some moment in the future.

Evidently, after the series of repairs the original schedule performance deteriorates. Hence, the schedule must be fully reconstructed at some moments of the production. The simplest approach is to reconstruct periodically after fixed time intervals. In a system with flexible dynamics, periodic rescheduling may not be optimal, because the reconstruction may be either too frequent or too rare. Alternatively, the reconstruction can be performed, when a severe disruption occurs. To not regress into the fully reactive method, the severity of the disruption must be assessed to decide the necessity of the reconstruction. For instance, authors of [41] employ a variant of the SVM to solve the assessment problem.

Overall, proactive reactive methods are intermediate between reactive scheduling and static scheduling which allows them to benefit from advances in both fields. Additionally, these methods benefit from robust scheduling methods due to additional resistance. For instance, authors of [42] employ PSO for multi-objective optimization to find robust schedules. The method is then employed in a proactive reactive framework for schedule construction.

1.9 Static Problem Benchmark

For JSSP and FJSSP evaluation and comparison, there are several sets of publicly available benchmarks. A single instance of the benchmark is represented by a set of 20-100 jobs available at time 0, each job has a sequence of operation processing times and a path through the shop floor usually consisting of 10-20 machines. In other words, these benchmarks focus on the standard

definition of the problem with the makespan minimization objective. For each instance, the lower and upper bounds of the makespan are available, which allows us to compare the relative performance of the method with the optimal solution. The most common benchmarks are the Tailard [43], DMU [44], and other benchmarks [45]. Since most of the methods usually operate in the extended definition of the problem, static problem benchmarks are only used to verify that the method preserves the ability to solve the standard problem.

1.10 Dynamic Problem Benchmarks

DJSSP is inherently closer to practical applications than JSSP. Depending on the application, the variety of the disruptions considered also varies. Hence, for each application, an individual benchmark instance must be developed. Yet, if we restrain from excessive disruption factors, the configuration of the common benchmark considers the following set of parameters [8, 46]:

- **Operation Processing Time:** The heterogeneity of the operation processing time is the main source of idleness in the JSSP problem. To model job processing times two probability distributions are often employed: uniform and normal. The uniform distribution is capable of creating various scenarios. The normal distribution is used in combination with uniform distribution to incorporate the stochasticity in job processing time. Depending on the requirements of the application, other probability distributions can be used. However, to preserve the transferability of the methods, the aforementioned variants are studied. In the context of the thesis, the mixture of the uniform and normal distribution is used. Let Uniform$[a, b]$ be the uniform distribution, $\mathcal{N}(0, \sigma^2)$ be the normal distribution of noise, $[n, m]$ be the boundary of the studied operation processing time. Then operation processing time of j-operation of job J_i on k-th machine ($p^j_{i,k}$) is sampled as follows:

$$\begin{aligned} t_1 &\sim \text{Uniform}[a, b] \\ t_2 &\sim \mathcal{N}(0, \sigma^2) \\ p_{i,k} &= \max(\min(t_1 + t_2, m), n) \end{aligned} \tag{1.2}$$

- **Time Till Due:** Time till due is applied to incorporate job priorities. The most common approach is to model it using the tightness factor λ, which is the multiplier of job total processing time. The tightness factor is usually sampled from the uniform distribution, the parameters of which depend on the specific application. However, it is worth mentioning that too tight or too loose due dates can lead to either infeasible or trivial solutions. It is usually recommended to make time till due at least 2 times the total processing time of the job [47]. In the context of the thesis,

we follow the approach from [8] which estimates total job processing time as the mean of operation processing times on each machine multiplied by the total number of job operations. The reason for such a definition is to incorporate heterogeneity of processing times from the FJSSP problem, while for the JSSP problem, the value is equivalent to the sum of operation processing times. Let Uniform$[a, b]$ be the uniform distribution, $\overline{p_i}$ be the mean of processing times of job J_i time till due is sampled as follows:

$$\begin{aligned} \lambda_i &\sim \text{Uniform}[a, b] \\ TTD_i &= NOW + (1 + \lambda_i) O_i \overline{p_i} \end{aligned} \tag{1.3}$$

- **Job Arrival:** Job arrival is modeled to correspond to the expected utilization rate of the system. Let $\mathbb{E}(t)$ be the expected operation processing time for all jobs, $\mathbb{E}$(interval) be the expected duration of intervals between job arrival, then the expected utilization rate $\mathbb{E}$(util) is defined as follows:

$$\mathbb{E}(\text{util}) = \frac{\mathbb{E}(t)}{\mathbb{E}(\text{interval})} * 100\% \tag{1.4}$$

From the previous equation, the expectation of expected interval lengths between job arrival is derived as follows:

$$\mathbb{E}(\text{interval}) = \frac{\mathbb{E}(t)}{\mathbb{E}(\text{util})} \tag{1.5}$$

Arrival intervals are usually modeled using the Poisson random process. Hence, the arrival time follows the exponential distribution, i.e. let μ be the floating point value for the utilization rate for the simulation, then the expected duration of intervals between job arrival a_i is computed as follows:

$$a_i \sim \text{Exp}(\frac{\mu}{\mathbb{E}(t)}) \tag{1.6}$$

- **Irregularities in the operation sequence:** The number of job operations and job paths through the shop floor can vary significantly depending on the practical application. The most restrictive approach is to consider that all jobs have $|\mathcal{M}|$ operations and pass through all machines in the same order (flow shop problem). Alternatively, we can consider that the job has a variable length and must visit machines in fully random order. In the scope of the thesis, we consider the latter. The job path is constructed by generating a random permutation of machines on the shop floor. The variability of length is achieved by removing a suffix of length k from the permutation where k is sampled from the uniform distribution.

- **Machine breakdowns and maintenance:** Machine breakdowns and maintenance are usually dependent on the specific application of the JSSP problem. Their models can include various factors, e.g. the residual life of the machine, average utilization rate, wear factor, the latest maintenance date, etc. [48]. While the model considering the aforementioned parameters can result in a more realistic understanding of the disruption impact and, hence, improve the quality of the model, the effect of the disruption is similar to any other kind of dynamic event. Since we are usually interested in robust solutions, machine breakdowns, and maintenance are usually modeled stochastically. In the scope of the thesis, the disruption arrival is modeled using the Poisson process running for every machine on the shop floor, and the duration of the disruption repair is modeled with uniform distribution.

- **Number of jobs on the shop floor:** Under the condition of minimizing the regular criterion the optimal schedule is usually obtained with a no-wait scheduling. If we consider that there is no job on the shop floor at the beginning of the simulation and job arrival times follow the utilization rate model mentioned earlier, we observe that a vast majority of machine decisions were made with only one job in the machine queue. We call such a decision *passive*, while the scheduling decision with more than one job in the machine queue is called *active*. To be more precise the ratio of active decisions to all decisions is proportional to the expected utilization rate and inversely proportional to the number of machines on the shop floor. For further details, we refer to Section 5.2.3 of [8]. While the issue can be mitigated by increasing the number of jobs available at the beginning of the simulation on the shop floor, the complexity of the problem is increased.

Chapter 2

Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that stands on the idea of learning an optimal behavior through continuous interactions with the environment. The inspiration for these methods usually comes from the way human beings learn to perform various tasks. For instance, learning to ride a bicycle involves a series of unsuccessful attempts, until the rider learns to maintain the balance. The learning curve doesn't stop after the first successful ride, but it extends until the driving becomes an automatic process. If we look at the process of learning, it notably happens without a teacher. The rider instinctively grasps the cause-and-effect relationship through repeated attempts. After the so-called exploration phase, the rider develops his understanding until he performs the optimal sequences of actions that lead to the desired outcome. This phase of learning is called exploitation.

The first major success of the RL method was TD-Gammon, a backgammon-playing program developed by Gerald Tesauro in 1992 [49]. The method achieved a level comparable to intermediate human players after learning from 1.5 million self-play games. While the method itself led to the creation of new strategies in backgammon, it wasn't able to beat experienced players. The supremacy of RL was achieved 5 years later when Deep Blue, a chess-playing program developed by IBM, defeated the reigning world champion Garry Kasparov in 1997 [50]. However, these methods were limited to games with simple state representation and usually required enormous computational resources. The breakthrough in the field of RL came in 2013 when DeepMind developed a method called Deep Q-Network [51], which combines RL with Convolutional Neural Networks to learn to play a collection of different Atari games. For the first time, the method based on RL and Neural Networks was able to outperform human players in three Atari games [51]. The goal of this chapter is to introduce the basic concepts of RL and briefly describe RL algorithms used in job shop scheduling problems. The notation used in the chapter is based on the book [52].

2.1 Exploration and Exploitation

Similarly to metaheuristics described in the previous chapter, RL methods learn through the process of exploration and exploitation. Generally speaking, the agent in RL methods learns to perform a sequence of actions to maximize the cumulative strength of the signal called *return*. To learn such a sequence the agent must explore the environment by trying different actions and observing the consequences of them (*exploration phase*). The agent receives the *response signal* from the environment called reward usually after each executed action. It is worth mentioning that the effect of the action on the return is neither immediate nor stable. For instance, one bad move in chess mid-game can lead to a loss of the game if the opportunity is taken by the opponent. The learning is usually performed by associating either the state or the state and action with the expected value of the return. The exploitation is then achieved by selecting such action that maximizes the expected value of return. To obtain the optimal rate of convergence and a good estimation of the return, the balance between exploration and exploitation is crucial. The way to achieve it usually depends on the method employed.

2.2 Markov Decision Process

Problems in RL are usually formulated as Markov Decision Processes (MDP) or their further extensions as presented in [53].

▶ **Definition 2.1** (Markov Decision Process)**.** *A Markov Decision Process is a 5-tuple* (S, A, P, ρ, γ) *where*

- S *is a set of states*
- A *is a set of actions*
- $p : S \times A \to \mathcal{P}(S) \times \mathbb{R}$ *is the transition model, with* $p(s_{t+1}, r_{t+1}|s_t, a_t)$ *being the probability of transitioning into state* $s_{t+1} \in S$ *from state* $s_t \in S$ *by taking action* $a_t \in A$ *and receiving reward* $r_{t+1} \in \mathbb{R}$
- ρ *is the initial distribution of states*
- $\gamma \in [0, 1]$ *is discount factor*

Contrary to the deterministic planning problem, the MDP transitions to the next state stochastically. The term "Markov" in the MDP refers to the Markov property from Markov Chain theory, which states that the next state and obtained reward depend only on the information of the current state and the action taken, which can be denoted in the following way:

$$p(s_{t+1}, r_{t+1}|s_t, a_t) = p(s_{t+1}, r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) \tag{2.1}$$

Reinforcement Learning methods commonly learn from an agent-environment interaction sequence called a *trajectory*.

▶ **Definition 2.2** (Trajectory)**.** *A trajectory is a sequence of states, actions, and rewards*

$$\tau = (s_1, a_1, r_2, s_2, a_2, r_3, \ldots, s_t, a_t, r_{t+1}, \ldots) \tag{2.2}$$

where

$$\begin{gathered} s_1 \sim \rho \\ \forall t \in T : a_t \sim \pi(a_t|s_t) \\ \forall t \in T : s_{t+1}, r_{t+1} \sim p(s_{t+1}, r_{t+1}|s_t, a_t) \end{gathered} \tag{2.3}$$

In some applications, the trajectory of agent-environment interactions can be naturally broken into subsequences called *episodes*. Each episode ends in a special state called the *terminal state* followed by the independent state usually sampled from the distribution of initial states ρ. For instance, in games, each playthrough can be considered an episode. The aforementioned task is called *episodic*. For episodic tasks, we consider a set of terminal states denoted as S^+. Given the episode starting in the state s_t and ending in the state $s_T \in S^+$, the return G_t, which we seek to maximize, is defined as the sum of obtained rewards.

$$G_t = r_{t+1} + r_{t+2} + \ldots + r_T \tag{2.4}$$

Alternatively, the agent-environment interactions sequence could go on continually without the terminal state. We call such tasks *continuous*. Note, that the return in the continuous task couldn't be defined as the sum of rewards, because the sum may be divergent due to the infinite length of the trajectory. Hence, the discount factor $\gamma \in [0, 1]$ is introduced. The *discounted return* is then defined as follows:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.5}$$

If the discount factor $\gamma < 1$, the sum converges. In general, the discount factor γ determines the importance of future rewards. If the discount factor is close to 0 the agent focuses on the immediate rewards, while the discount factor close to 1 enforces the agent to take into account future rewards.

Most of the RL methods can be applied to both episodic and continuous tasks. To define the return for both tasks, we consider that the episode in episodic tasks is represented as an infinite sequence in the trajectory. When an agent achieves the terminal state, it is absorbed in it and always receives the reward of 0. In other words, the trajectory is represented as the concatenation of infinite sequences for each episode, i.e.

$$\begin{aligned}\tau =&(s_{1,1}, a_{1,1}, r_{2,1}, \ldots, s_{T,1}, a_{T,1}, r_{T,1}, s_{T,1}, a_{T+1,1}, 0, s_{T,1}, a_{T+2,1}, 0 \ldots) \;||\\ &(s_{1,2}, a_{1,2}, r_{2,2}, \ldots, s_{T,2}, a_{T,2}, r_{T,2}, s_{T,2}, a_{T+1,2}, 0, s_{T+2,2}, a_{T+2,2}, 0 \ldots) \;||\\ &\ldots\end{aligned} \tag{2.6}$$

where $||$ stands for the concatenation and the additional index denotes the episode number. The return at state t is then defined as follows

▶ **Definition 2.3** (Return)**.** *Let τ be the trajectory, γ be the discount factor and T be the final time step of the information. The return at state $s_t, t < T$ is defined when $T \neq \infty$ or $\gamma < 1$ as*

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \tag{2.7}$$

The restriction $T \neq \infty$ or $\gamma < 1$ in the definition ensures that the return is bounded. In case T is finite, the return is finite, because rewards are finite. Otherwise, the $\gamma < 1$ ensures the convergence of the infinite sum. Finally, we would note that returns at successive time steps are related to each other, i.e.

$$\begin{aligned}G_t &\doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \\ &= r_{t+1} + \gamma \left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \cdots \right) \\ &= r_{t+1} + \gamma G_{t+1}\end{aligned} \tag{2.8}$$

The behavior of the agent is defined by the policy $\pi(a|s)$ which is the mapping from the environment state to the probability distribution over actions $(S \to A)$. Let's for now consider that we have obtained some policy π. Almost all RL methods estimate either the quality (in terms of the expected return) of the agent being in the particular state or the quality of the agent taking the particular action in the particular state. The estimation is usually done by either the value function or the state-action function correspondingly.

▶ **Definition 2.4** (Value Function)**.** *Let π be the policy and $s \in S$ be the state. We define the value function of state π under policy π as*

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

▶ **Definition 2.5** (Action-Value Function)**.** *Let π be the policy, $s \in S$ be the state and $a \in A$ be the action. We define the action-value function of taking action a in state s under policy π as*

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right]$$

Both the value function and the action-value functions represent the expected return, hence, one is expressable in terms of another and vice versa.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_{a\sim\pi}[q_\pi(s,a)] \\ q_\pi(s,a) &= \mathbb{E}_{s',r\sim P}[r + \gamma v_\pi(s')] \end{aligned} \tag{2.9}$$

In equation 2.8 the recursive relationship between return at step t and $t+1$ was shown. A similar recursive relationship can be shown for the value function for all $s \in S$.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t \mid s_t = s\right] \\ &= \mathbb{E}_\pi\left[r_{t+1} + \gamma G_{t+1} \mid s_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p\left(s', r \mid s, a\right)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid s_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right] \end{aligned} \tag{2.10}$$

The final equation in 2.10 can be read as the expected value of random variable $r_{t+1} + \gamma G_{t+1}$ with the probability of $\pi(a|s)p(s', r|s, a)$. The equation is also known as the Bellman equation for the value function [54].

▶ **Definition 2.6** (Bellman Equation for Value Function)**.** *Let π be the policy, $s \in S$ be the state, $a \in A$ be the action, and $r \in \mathbb{R}$ reward. Then the Bellman equation for the value function v_π is defined as*

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid s_t = s\right] = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s' \in S, r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right]$$

The goal of the RL methods is to find the policy π^* that maximizes the expected return. Intuitively, the policy π is better than the policy π' if the expected return of the policy π is greater than or equal to the expected return of the policy π' for all states $s \in S$. In other words, π is better than π' if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$, because the value function is exactly the expected return in the state under the policy π. There is always at least one policy that is better or equal to all other policies. All such policies are called *optimal* and are denoted as π_*. The value function of the optimal policy is called the optimal value function and is denoted as v_*.

▶ **Definition 2.7** (Optimal Value Function)**.** *The optimal value function v_* for all states $s \in S$ is defined as*

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{2.11}$$

Optimal policies share the same optimal action-value function q_*.

▶ **Definition 2.8** (Optimal Action-Value Function)**.** *The optimal action-value function q_* for all states $s \in S$ and actions $a \in A$ is defined as*

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \tag{2.12}$$

Since the v_* is the value function for a policy, it must satisfy the Bellman equation for the value function. However, due to the optimality of the policy, it can be expressed without it, i.e.

$$\begin{aligned} v_*(s) &= \max_a q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}\left[G_t \mid S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_*}\left[r_{t+1} + \gamma G_{t+1} \mid s_t = s, a_t = a\right] \\ &= \max_a \mathbb{E}\left[r_{t+1} + \gamma v_*\left(S_{t+1}\right) \mid s_t = s, a_t = a\right] \\ &= \max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_*\left(s'\right)\right]. \end{aligned} \tag{2.13}$$

The equation is the Bellman optimality equation for the value function, which states the fact that the return in the state s under the optimal policy is equal to the expected return by taking the best action from that state. The Bellman optimality equation for the action-value function is the following.

$$\begin{aligned} q_*(s,a) &= \mathbb{E}_{s' \sim P}\left[R_{t+1} + \gamma \max_{a'} q_*\left(S_{t+1}, a'\right)\right] \\ &= \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \max_{a'} q_*\left(s', a'\right)\right]. \end{aligned} \tag{2.14}$$

The Bellman Equations are the basic building blocks of various RL algorithms. In the case of the finite MDP with a finite number of states and actions, we can use algorithms like *Value Iteration* and *Policy Iteration* to find the optimal policy. Some algorithms additionally define an advantage function which represents the change in the value function by taking the action a in the state s.

▶ **Definition 2.9** (Advantage Function)**.** *Let π be the policy, $s \in S$ be the state and $a \in A$ be the action. The advantage function is defined as*

$$a_\pi(s,a) = q_\pi(s,a) - v_\pi(s) \tag{2.15}$$

2.3 Q-learning

The goal of the RL methods is to employ value functions in order to guide the search for the optimal policy. Since the value functions are unknown, their values are estimated through the agent-environment interaction. The core method in RL to estimate the value function under policy π is the Temporal

Difference (TD) learning. The TD learning performs the update of the estimate after k interactions with the environment. The simplest update of the TD learning would happen after one interaction and is the following:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right] \tag{2.16}$$

where $V(s_t)$ is the estimated value of the value function for state s_t and α is the learning rate. Note, that the update is performed based on the current estimate. In other words, the method updates the guess based on the previous guess, i.e. it *bootstraps*. The quantity in the brackets represents the error between the current estimate $V(s_t)$ and the new estimate $r_{t+1} + \gamma V(s_{t+1})$. This quantity is called the *TD error* and it is usually denoted in the following way:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.17}$$

After obtaining a good estimate, the policy can be easily derived by selecting the action with the highest value. The aforementioned behavior of the agent is called *greedy* and it is suitable for the evaluation of the policy. However, the greedy policy estimation limits the exploration of the environment. Hence, one of the ways to control the exploration is to employ ϵ-greedy policy, i.e. the policy takes greedy action with the probability $1 - \epsilon$ and a random action with the probability ϵ. The former methods that employ the policy derived from the current estimation of value functions are called *on-policy* methods, while the latter methods that take actions different from the current estimation of value functions are called *off-policy* methods.

For the following section, we suppose that the set of states S and the set of actions A in the MDP are finite. The methods that operate in the finite MDP are called *tabular*. The common training procedure of the tabular TD method is presented in Algorithm 1. In the algorithm, we suppose that policy includes all possible modifications, i.e. ϵ-greedy, etc. The procedure for estimating the action-value function is derived by replacing the value table with the action-value table.

The Q-learning [55] is the off-policy tabular TD-Learning method with the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.18}$$

Q-learning method supposes that the new estimate of the action-value function is the discounted maximum over current estimates and the reward. The problem with such an approach is that the maximum of estimated values can lead to significant positive bias. For example, we consider the MDP presented in Figure 2.1 taken from [52]. In initial state A the agent can either go to terminal state B or to state C. In both cases, it receives the reward of 0. From the state C the agent has many actions all leading to the terminal state D

Algorithm 1: Tabular TD Learning

Input: The policy to be evaluated π, the learning rate α
Output: Estimated value function V

Initialize arbitraly $V(s)$ for all $s \in S$ and $V(s_{\text{terminal}}) = 0$
while *the termination condition is not met* **do**
  $S \leftarrow$ Initial state sampled from ρ
  while S *is not terminal* **do**
    A $\leftarrow$ action given by π for state S'
    Take action A and observe R, S'
    $\delta \leftarrow R + \gamma V(S') - V(S)$
    $V(S) \leftarrow V(S) + \alpha\delta$
    $S \leftarrow S'$
  end
end

with the reward sampled from the normal distribution with the mean of -0.1. The expected return for the transition from state A to state C is -0.1, while the expected return for the transition from state A to state B is 0. Hence, the optimal policy is to go to state B. However, the Q-learning algorithm may favor the transition from state A to state C, if the expected return of the transition happens to be greater than 0. In the end, the estimation converges to the true value of -0.1, but it significantly slows down the convergence of the method. This phenomenon is called the *maximization bias*.

One of the ways to overcome the issue is to use the Double Q-learning algorithm [56]. To minimize the bias the algorithm updates two estimates of action value function $Q_1(s, a)$ and $Q_2(s, a)$. The first action value function estimate $Q_1(s, a)$ is used to select the action maximizing the expected return $A^* = \arg\max_a Q_1(s, a)$, while the second one $Q_2(s, a)$ provides the estimate of its value $Q_2(s, A^*) = Q_2(s, \arg\max_a Q_1(s, a))$. The estimate is unbiased, i.e. $\mathbb{E}[Q_2(s, A^*)] = q(s, A^*)$. This leads to the following update procedure for $Q_1(s, a)$:

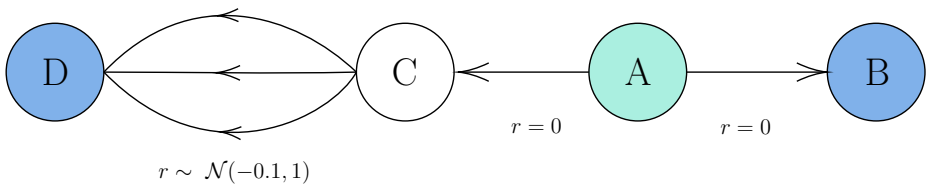


Figure 2.1 MDP for maximization bias example. The state A is the initial state and states B and D are terminal states.

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q_2(s_{t+1}, \text{argmax}_a \, Q_1(s_{t+1}, a)) - Q_1(s_t, a_t) \right] \tag{2.19}$$

Similar procedure can be applied to $Q_2(s, a)$, i.e.

$$Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q_1(s_{t+1}, \text{argmax}_a \, Q_2(s_{t+1}, a)) - Q_2(s_t, a_t) \right] \tag{2.20}$$

At each step, the method only updates one estimate of the action-value function, which is selected randomly. Finally, the estimation of the action-value function for policy construction is derived w.r.t. both estimates, for instance, as the sum, i.e. $Q(s, a) = Q_1(s, a) + Q_2(s, a)$.

2.4 Deep Q-Learning

The main limitation of the Q-learning algorithm is the finite size of the MDP state space. One of the ways to solve the problem is to learn the approximate action-value function. However, when function approximation is integrated into the Q-learning algorithm, the obtained method may diverge during the training. It is because, we have created the combination known as the *deadly triad*, i.e. function approximation, bootstrapping, and off-policy learning. The Baird counterexample [57] shows the training divergence of the deadly triad. The problem may be avoided by giving up one of the three components, but it yields an increase in the computational complexity.

The main breakthrough had been achieved by the Deep Q-Network (DQN) paper [51]. The paper comes with two significant improvements: target network and experience replay buffer. The first improvement is the idea of the target network, which represents a copy of the original network with frozen parameters used to estimate the target value. During the training, the parameters of the current action value function estimate are copied to the target network usually after a fixed number of training steps. Note, that the value function is still bootstrapped, but the delay in the update of the target network helps to break the conditions of the deadly triad. The second improvement is the experience replay buffer, i.e. buffer for storing agent-environment interactions for training. Since Q-learning requires only the current and next state, action, and reward for learning, the experience from exploration can be used multiple times. This usually leads to a more stable training process. The DQN algorithm is presented in Algorithm 3.

For the DQN algorithm, many improvements have been proposed. The combination of the most successful improvements has led to a new architecture called Rainbow [58]. The Rainbow architecture combines the following improvements: double Q-learning, dueling network, prioritized experience replay, noisy networks, multi-step learning, and distributional RL. In the scope of the thesis, we study the first 5 improvements of the Rainbow architecture.

Algorithm 2: Deep Q-Network

Input: Q-function parameters θ, experience replay memory M, target update interval C

Set the target network parameters $\theta_{\text{target}} \leftarrow \theta$

while *the termination condition is not met* **do**

  Observe state S, select action A according to ϵ-greedy policy, i.e.

  $$A = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \text{argmax}_{A'} Q(S, A'; \theta), & \text{otherwise.} \end{cases}$$

  Execute action A, observe reward R, next state S' and done flag D

  Store transition (S, A, R, S', D) in M

  Sample minibatch $B = \{(S, A, R, S', D)\}$ from M

  Compute new targets for each transition $(S_i, A_i, R_i, S'_i, D_i) \in B$

  $$y_i = R_i + \gamma(1 - D_i) \max_{A'} Q(S'_i, A'; \theta_{target})$$

  Perform a gradient step w.r.t. θ on

  $$\sum_{i=0}^{|B|} (y_i - Q(S_i, A_i; \theta))^2$$

  Every C steps reset target network

  $$\theta_{\text{target}} \leftarrow \theta$$

end

The DQN algorithm similar to Q-learning suffers from the maximization bias. Similarly, to the Double Q-learning, the double DQN (DDQN) algorithm [59] employs two action-value function estimates. However, instead of learning a second estimate the target network is used. This leads to the following loss for the DDQN:

$$(R + \gamma(1 - D)Q(S', \text{argmax}_{A'} Q(S', A'; \theta); \theta_{\text{target}}) - Q(S', A; \theta))^2 \tag{2.21}$$

The dueling network architecture is the neural network architecture designed for value function estimation [60]. It splits the network into two streams, one for the state value estimation and the other for the advantage function estimation sharing the same encoder. The final value function is then derived as the sum of the state value and the advantage function, i.e.

$$Q(s, a; \epsilon, \phi, \nu) = V(f(s; \epsilon); \phi) + A(f(s; \epsilon), a; \nu) - \frac{1}{N_{\text{Actions}}} \sum_{a'} A(f(s; \epsilon), a'; \nu) \tag{2.22}$$

where ϵ, ϕ, and ν are the parameters of the encoder, the value stream, and the advantage stream respectively.

The DQN algorithm samples uniformly from the experience replay buffer. However, for large buffers, the uniform sampling may lead to the inefficient use of the experience. Ideally, the agent should sample more frequently transitions that have a lot to learn from, i.e. have a high TD error. A prioritized experience replay [61] buffer attempts to solve the issue by sampling transitions with the probability proportional to the absolute value of TD error, i.e.

$$p_t \propto \left| R + \gamma(1 - D) \max_{A'} Q(S', A'; \theta_{target}) - Q(S, A; \theta) \right|^{\omega} \tag{2.23}$$

where w is the hyperparameter that determines the shape of the distribution. New transitions are added to the buffer with the maximum priority.

The ϵ-greedy policy can significantly limit the exploration of the environment when a lot of actions must be taken to receive the reward. Noisy networks [62] propose a noisy linear layer that combines deterministic and noisy behavior, i.e.

$$\boldsymbol{y} = (\boldsymbol{b} + \mathbf{W}\boldsymbol{x}) + \left(\boldsymbol{b}_{\text{noisy}} \odot \epsilon^b + (\mathbf{W}_{\text{noisy}} \odot \epsilon^w)\, \boldsymbol{x} \right) \tag{2.24}$$

where ϵ^b and ϵ^w are random noise variables and $\odot$ denotes the element-wise multiplication. The introduction of noisy nets transforms DQN from off-policy to on-policy. Over the training, the algorithm can learn to ignore the noise, but this happens at different rates in different parts of the state space.

Rewards obtained after a single step can be noisy and may not represent the effect of the action. Alternatively, the multi-step learning algorithm [63] proposes to estimate the return by forward-view targets. The truncated return is then estimated as follows:

$$G_{t:t+n} = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} \tag{2.25}$$

The training is done w.r.t. the following loss:

$$\left(G_{t:t+n} + \gamma^n \max_a Q(S_{t+n}, a; \theta) - Q(S_t, A_t; \theta) \right)^2 \tag{2.26}$$

2.5 REINFORCE

Instead of approximating the value function, the agent can directly learn a stochastic parametrized policy. To explore the environment the suitable action is sampled from the current estimation of the policy, i.e. the method is on-policy. When the agent is confident in the specific action, the action has a high probability of being selected, so the exploitation is achieved. One of the ways to train the parametrized policy is to use the gradient descent method or

its extension. Our goal is to maximize the expected return, i.e. the following objective function

$$J(\theta) = v_{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}[G_t|s_0] \tag{2.27}$$

where s_0 represents an initial state. However, to compute the expected value of return, the state distribution at each step of agent-environment interaction is needed, which is the function of the environment dynamics and is usually unknown. One of the ways to overcome the issue is to use the Policy Gradient Theorem.

▶ **Theorem 2.10** (Policy Gradient Theorem [64]). *Let τ be the trajectory. It holds*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\Psi_t\frac{\nabla_\theta\pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\Psi_t\nabla_\theta\log\pi_\theta(a_t|s_t)\right] \tag{2.28}$$

where Ψ_t is any of the following:

- $\Psi_t = G_t$*: the return*
- $\Psi_t = G_t - b(s_t)$*: the return with baseline*
- $\Psi_t = Q_{\pi_\theta}(s_t, a_t)$*: the action-value function*
- $\Psi_t = A_{\pi_\theta}(s_t, a_t)$*: the advantage function*

For truncated trajectory, the gradient can be estimated and it is proportional to the exact value of the gradient, i.e.

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t}^{T}\Psi_t\nabla_\theta\log\pi_\theta(a_t|s_t)\right] \tag{2.29}$$

where T is the final step of the truncated trajectory.

The policy is a function $f : S \to A$ and the gradient through it can be computed by the backpropagation algorithm implemented by all deep learning frameworks. The Policy Gradient Theorem states that the sum of states weighted by the return over the probability of action given by the policy in a given state is proportional to the exact policy gradient, i.e. differs by constant scale factor. Since there are no constraints on the learning rate, the scale of the gradient is absorbed by it. The detailed proof of the theorem can be found in [52]. In practice, the gradient is computed over a sample of records of limited size. Hence, the tradeoff between the reasonable size of the sample and the quality (in terms of variance) of the estimated gradient is crucial. It is the main reason why in the definition of the theorem there are several forms of the

Algorithm 3: REINFORCE with Baseline

Input: Parametrized policy $\pi(a|s;\theta)$, parameterized value function $v(s;\psi)$, policy learning rate α, value function learning rate β

while *the termination condition is not met* **do**

Generate an episode $\tau = (S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_T)$ using $\pi(a|s;\theta)$

Compute the return for each state s_t

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

Take the gradient step w.r.t. θ on

$$\sum_{t=0}^{T} (G_t - v(s_t;\psi)) \nabla_\theta \log \pi(A_t|S_t;\theta)$$

Take the gradient step w.r.t. ψ on

$$\sum_{t=0}^{T} (G_t - v(s_t;\psi))^2$$

end

Ψ_t. It can be shown that the estimation with return $\Psi_t = G_t$ has the highest variance, while the estimation with the advantage function $\Psi_t = A_{\pi_\theta}(s_t, a_t)$ is low-variance estimator [65].

REINFORCE algorithm [66] is the policy gradient method that uses the return as the Ψ_t. However, the return is a high-variance estimator, because it is just the sum of rewards that depends on the policy. One of the ways to reduce the variance is to train a baseline function $b(s), s \in S$ that reduces the variance of the return. Note, that the baseline is a function of the state, which means it doesn't affect the policy function gradient. A suitable choice for the baseline function can be the value function, i.e. $b(s_t) = v_{\pi_\theta}(s_t)$ [67]. A better choice for the baseline function can be the advantage function, i.e. $b(s_t) = A_{\pi_\theta}(s_t, a_t)$. Overall, this leads to the Algorithm 3.

2.6 Proximal Policy Optimization

The policy gradient methods are first-order optimization methods. They are capable of estimating the direction of the gradient but not the step size. This usually leads to either too large or too small steps during optimization, which results in policy forgetting or slow convergence respectively. The problem

has led to the development of the Policy Gradient extension called Natural Policy Gradient (NPT) [68]. One of the NPT methods is Trust Region Policy Optimization (TRPO) [69], which constitutes the following optimization task

$$\begin{aligned} &\text{maximize}_\theta \, \mathbb{E}_{s\sim\rho(\pi_{\theta_0}),a\sim\pi_{\theta_0}} \left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_0}(a \mid s)} A_{\pi_{\theta_0}}(s,a)\right] \\ &\text{subject to } \mathbb{E}_{s\sim\rho(\pi_{\theta_0})}[D_{\text{KL}}\left(\pi_{\theta_0} \| \pi_\theta\right)] < \delta \end{aligned} \tag{2.30}$$

where π_{θ_0} is the initial policy, $\rho(\pi_{\theta_0})$ is the state distribution under the initial policy, $A_{\pi_{\theta_0}}(s,a)$ is the advantage function, and $D_{\text{KL}}\left(\pi_{\theta_0} \| \pi_\theta\right)$ is the Kullback-Leibler divergence between the initial and the new policy, δ is improvement constant. Although the in-depth explanation of the TRPO is out of the thesis's scope, the details of the optimization task are essential for further discussion. In general, the agent must maximize the advantage, which states if the action leads to the increase in return ($A(s,a) > 0$) or not ($A(s,a) < 0$). Hence, the new policy should have increased the probability of actions with positive advantage and decreased the probability of actions with negative advantage. At the same time, the step in the direction of the improvement shouldn't be excessively large to avoid policy collapse, i.e. the maximum size of the step is δ. δ is the constant and it is found empirically for the specific problem. Finally, to compute the length of the step the KL-divergence between the old and a new policy is used.

The TRPO is the second-order optimization method, i.e. at some step the optimization procedure involves the computation of the Hessian w.r.t. policy parameters. Proximal Policy Optimization (PPO) [70] method is a popular simplification of TRPO that converts the algorithm to the first-order optimization task while preserving the performance of TRPO. The method proposes a modified TRPO objective that penalizes significant changes in the policy, i.e.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{2.31}$$

where ϵ is policy change step size, $\hat{A}_t$ is a advantage function estimate and $r_t(\theta)$ is the policy update ratio, i.e.

$$r_t(\theta) = \frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_0}\left(a_t \mid s_t\right)} \tag{2.32}$$

The advantage is estimated with the generalized advantage estimation (GAE) [64] that is defined as follows:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.33}$$

where λ is the eligibility trace parameter and δ_t is the TD error at moment t, T is the final step of the truncated trajectory.

Algorithm 4: Proximal Policy Optimization

```
for iteration i = 1, 2, ... do
    for actor j = 1, 2, ... do
        Run policy π_θ0 in environment for T timesteps
        Compute advantages estimates Â_1, ..., Â_T
    end
    Optimize surrogate objective w.r.t. θ for K epochs and minibatch
     size M
    θ_0 ← θ
end
```

The neural network for the PPO algorithm commonly follows an actor-critic architecture, i.e. the actor represents the parametrized policy and the critic represents the parametrized value function. One of the properties the method possesses is the fast convergence that leads to limited exploration of the environment. To motivate the exploration of the environment the loss is augmented with the discrete entropy of π_θ term. The final objective that the method maximizes is the combination of the clipped surrogate objective, the entropy, and the value function loss, i.e.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{2.34}$$

where c_1 is value loss coefficient, c_2 is the entropy coefficient, $S[\pi_\theta](s_t)$ is the discrete entropy of policy at state s_t. The training procedure of the PPO algorithm is presented in Algorithm 4.

The reader shouldn't be misled by the simplicity of the pseudo-code. The method possesses high sensitivity to the hyperparameters, i.e. the learning rate, number of epochs, and batch size as well as the initialization of the neural network. At the same time, the implementation usually involves a series of tricks e.g. gradient normalization, advantage normalization, entropy regularization coefficient decay, etc. [71]. To analyze the training process, the development of policy entropy should be monitored [72].

Chapter 3

Graph Neural Networks

From the data-modeling perspective, a natural representation of the system state is a disjunctive graph discussed in Chapter 1. The goal of this chapter is to introduce the reader to the basic principles of graph neural networks as well as the graph embedder methods used in the thesis.

3.1 Message Passing Paradigm

Message Passing Paradigm (MPP) is a general framework for designing graph neural network embedders. The MPP can be thought of as an extension of the convolution operator that computes the value from the data local to the node. Let $G = (V, E)$ be a graph with the set of nodes V and the set of edges E. The most common definition of the MPP is the following [73]:

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \bigoplus_{j \in \mathcal{N}(i)} \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{i,j} \right) \right) \tag{3.1}$$

where

- $\mathbf{x}_i^{(k)} \in \mathbb{R}^n$ is the representation of the node i of dimensionality n at the k-th layer or the *message*
- $\mathbf{e}_{i,j} \in \mathbb{R}^d$ is the representation of the edge features between nodes i and j of the dimensionality d
- $\mathcal{N}(i)$ is the neighbourhood of the node i
- $\phi^{(k)}$ is the encoder function of node-edge features at layer k e.g. a neural network
- $\gamma^{(k)}$ is the composition function of current node features and hidden representation of the neighborhood at layer k, e.g. a neural network

- $\bigoplus$ is a permutation-invariant function, e.g. sum, mean, max, etc.

The output of an MPP graph neural network is a latent representation of the node applicable to a large variety of tasks such as node classification, link prediction, and graph classification. In node classification, the latent representation of the node is directly used to classify the node. In link prediction, the existence of an edge is modeled as the similarity criterion between a pair of nodes e.g. distance-based metric, sigmoid, etc. Finally, in graph classification, the most common approach to obtain the graph representation is to pass the latent representation of the nodes through *readout* or *pooling* functions. The simplest variant of a readout function is permutation-invariant pooling functions such as sum, mean, or max taken across the latent representation of all graph nodes. While these functions are simple, they may lack the representative power to capture the structure of the graph. Recently, there has been a growing interest in more sophisticated readout functions such as Hierarchical Graph Pooling, Memory Readout, etc. [74].

From the theoretical perspective, the representative power of Graph Neural Networks (GNN) is analyzed by comparing the GNN with the Weisfeiler-Lehman graph isomorphism test [75]. The task of determining the graph isomorphism between a pair of graphs belongs to the class of NP-intermediate problems, which means no polynomial-time algorithm is known to solve the problem. The Weisfeiler-Lehman test assigns every node of the graph a label. The output of the test are sets of node labels for the graph in the tested pair. If sets are distinct for a pair, then the graphs aren't isomorphic. Otherwise, the test doesn't tell us anything about the isomorphism of the graphs.

3.2 Graph Convolutional Networks

First, we introduce the Graph Convolutional Networks (GCN) [76]. The GCN uses the following update rule:

$$\mathbf{X}^{(k)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}^{(k-1)}\mathbf{W}^{(k-1)}\right) \tag{3.2}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ is the adjacency matrix of an undirected graph G with self-loops, $\tilde{\mathbf{D}}$ is the diagonal degree matrix of $\tilde{\mathbf{A}}$, $\mathbf{X}^{(k)} \in \mathbb{R}^{|V|\times d}$ is the node feature matrix at layer k, $\mathbf{W}^{(k-1)}$ is the weight matrix at layer $l-1$, and σ is the activation function e.g. ReLU, Sigmoid, etc. The equation 3.2 is equivalent to aggregating the neighborhood of every node in the graph. With the increasing number of layers k, the GCN aggregates the information from the further neighborhood of the node. In most cases, though, these networks are inefficient in aggregating neighborhood information, hence, in practice, the GCN is limited to a small number of layers. One of the attempts to improve the GCN and layers that we discuss in the next sections is the Jumping Knowledge

Algorithm 5: Graph Sample and Aggregate

Input: Graph $G = (V, E)$, node features $\mathbf{x}_v, \forall v \in V$, depth K, number of samples S, differentiable aggregation function AGGREGATE_k for all $k = 1, \ldots, K$, activation function σ, weight matrices $\mathbf{W}^k, \forall k = 1, \ldots, K$

Output: Node latent representation $\mathbf{z}_v, \forall v \in V$

$\mathbf{h}_v^{(0)} \leftarrow \mathbf{x}_v, \forall v \in V$
for $k = 1$ **to** K **do**
  for $v \in V$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^{(k)} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{(k-1)}, \forall u \in \mathcal{N}(v)\})$
    $\mathbf{h}_v^{(k)} \leftarrow \sigma\left(\mathbf{W}^{(k)} \cdot \text{CONCAT}\left(\mathbf{h}_v^{(k-1)}, \mathbf{h}_{\mathcal{N}(v)}^{(k)}\right)\right)$
  end
  $\mathbf{h}_v^{(k)} \leftarrow \mathbf{h}_v^{(k)} / \left\|\mathbf{h}_v^{(k)}\right\|_2, \forall v \in V$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^{(K)}, \forall v \in V$

(JK) that merges latent representations of nodes across all graph layers to increase awareness of the graph structure [77], i.e.

$$\mathbf{h}_G = \Big\|_{k=1}^{K} \left(\text{READOUT}\left(\left\{\mathbf{h}_j^{(k)} \mid v_j \in G\right\}\right) \mid k = 0, 1, \ldots, K\right) \tag{3.3}$$

where K is the depth of the number of GNN layers. Note, that the GCN is defined mainly for the undirected graphs, to apply it to the directed graph the adjacency matrix should be normalized only from the left side [78]. Overall, GCN represents one of the first successful attempts at implementing a deep graph convolution operator. At the same time, the GCN has some limitations such as weak generalization to the unseen nodes and high memory requirements for the large graphs [76].

3.3 Graph Sample and Aggregate

GCN performs well in the transductive setting where both the graph structure and node features are known beforehand. To be more precise, in the transductive setting, the subgraphs of the original graph are used for training, validation, and testing. However, in most cases, the method must operate well in the inductive setting, in which the network must generalize to unobserved data. As an attempt to address this issue, the Graph Sample and Aggregate (GraphSAGE) method was proposed [79]. The method follows the procedure shown in 5. Additionally, the authors consider a fixed-size neighborhood usually sampled from the graph to preserve constant computational complexity.

For the AGGREGATE function, the authors propose either taking the mean of the neighborhood and node itself or encoding with the bidirectional Long-Short Term Memory network [80] Alternatively, pooling methods discussed in the first section can be used, however the aggregation happens after the activation function. The original method was trained in a fully unsupervised manner with a negative sampling of distant nodes and a contrastive loss function. However, there are no restrictions to using GraphSAGE in the supervised setting.

3.4 Graph Attention Networks

The Graph Attention Networks (GAT) [81] employ a self-attention mechanism [82] to improve the aggregation of the neighborhood information by assigning different importance weights to the nodes. Given the node features $\mathbf{h}_i \in \mathbf{R}^d, v_i \in V$ the GAT computes the attention coefficients α_{ij} for all nodes in the neighborhood $v_j \in \mathcal{N}(v_i)$ of the node v_i as follows:

$$\mathbf{e}(\mathbf{h}_i, \mathbf{h}_j) = \sigma\left(\mathbf{a}[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]\right) \tag{3.4}$$

$$\alpha_{ij} = \frac{\exp\left(\mathbf{e}(\mathbf{h}_i, \mathbf{h}_j)\right)}{\sum_{k \in \mathcal{N}(v_i)} \exp\left(\mathbf{e}(\mathbf{h}_i, \mathbf{h}_k)\right)} \tag{3.5}$$

where $\mathbf{e}(\mathbf{h}_i, \mathbf{h}_j)$ is scoring function for edge between nodes i and j, $\mathbf{W} \in \mathbb{R}^{N,d}$ is the weight matrix of linear layer, $\mathbf{a} \in \mathbb{R}^{2N}$ is attention weight vector, and σ is the activation function. In their paper, the authors use LeakyReLU [83] with the slope of $\alpha = 0.2$ as the activation function. After obtaining the attention coefficients, the hidden representation of the node v_i is derived as a weighted sum of the neighborhood nodes' hidden representations, i.e.

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}(v_i)} \alpha_{ij} \mathbf{W}\mathbf{h}_j\right) \tag{3.6}$$

Finally, the GAT employs a multi-head attention mechanism to ensure the stability of attention weights training. Let K be the number of attention heads, then the hidden representation of the node given by the GAT layer is the following:

$$\mathbf{h}'_i = \Big\|_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_(v_i)} \alpha_{ij}^{(k)} \mathbf{W}^{(k)} \mathbf{h}_j\right) \tag{3.7}$$

On the final layer, the concatenation is no longer reasonable, hence, the values of attention heads are averaged, i.e.

$$\mathbf{h}'_i = \sigma\left(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in \mathcal{N}(v_i)} \alpha_{ij}^{(k)} \mathbf{W}^{(k)} \mathbf{h}_j\right) \tag{3.8}$$

Later, one important correction was proposed to the GAT. The authors of [84] propose the following correction to the original score function:

$$\begin{aligned} \text{GAT: } \mathbf{e}(\mathbf{h}_i, \mathbf{h}_j) &= \sigma\left(\mathbf{a}[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]\right) \\ \text{GATv2: } \mathbf{e}(\mathbf{h}_i, \mathbf{h}_j) &= \mathbf{a}\sigma\left(\mathbf{W}[\mathbf{h}_i \| \mathbf{h}_j]\right) \end{aligned} \tag{3.9}$$

They motivate the update by noting that the original GAT computes *static* attention. In terms of GNN, the attention mechanism is static if it always weighs one node at least as much as any other node of the graph unconditioned on the node generating the neighborhood. The static attention phenomenon can be spotted by looking at the attention coefficients matrix $\mathbf{A} \in \mathbb{R}^{|V|,|V|}$, in which attention scores of node j are larger in comparison to all other nodes. Static attention has limited expressiveness because it always focuses on one particular node from the neighborhood for every node in the graph. On the other hand, the correction makes GAT compute *dynamic* attention, which can focus on every node from the neighborhood and assign them different relevance scores. The dynamic attention mechanism possesses a higher level of expressiveness than the static one, which results in better model performance. The corrected version of the GAT is usually denoted as GATv2.

3.5 Graph Isomorphism Networks

Graph Isomorphism Networks [85] are proved to have the same representative power as the Weisfeiler-Lehman test in the "upper" bound. The main theorem stated in the original paper is the following:

▶ **Theorem 3.1.** *Let $\mathcal{A} : \mathcal{G} \to \mathbb{R}^d$ be a GNN. With a sufficient number of GNN layers, $\mathcal{A}$ maps any graphs G_1 and G_2 that the Weisfeiler-Lehman test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:*

- *$\mathcal{A}$ aggregates and updates node features iteratively with*

$$\mathbf{h}_i^{(k)} = \phi\left(\mathbf{h}_i^{(k-1)}, f\left(\left\{\mathbf{h}_j^{(k-1)} : v_j \in \mathcal{N}(v_i)\right\}\right)\right), \tag{3.10}$$

  where the functions f, which operates on multisets, and ϕ are injective.

- *$\mathcal{A}$'s graph-level readout, which operates on the multiset of node features $\left\{\mathbf{h}_i^k\right\}$, is injective.*

The reader can find proof of the theorem in the original paper. The injectiveness of the aggregate and readout functions over multisets (i.e. set with repeated elements) is the main requirement for the GNN to reach the expressive power of the WL-test. For the aggregate function, the paper proves that

the sum of node features can represent an injective function over a multiset which results in the following definition of the GIN layer:

$$\mathbf{h}_i^{(k)} = \phi\left(\left(1+\epsilon^{(k)}\right)\cdot \mathbf{h}_j^{(k-1)} + \sum_{v_j \in \mathcal{N}(v_i)} \mathbf{h}_j^{(k-1)}\right) \tag{3.11}$$

where ϕ is the Multi-Layer Perceptron (MLP), $\epsilon^{(k)}$ is the weight of the node generating the neighborhood that can be either learnable or constant. It must be noted, that there can be infinitely as many variants of GNN that possess the aforementioned property, but GIN is the simplest architecture from a design perspective. At the same time, GIN focuses specifically on the representative power of the GNN, while for some practical applications as mentioned in the paper, the information associated with the node may have higher importance for the task. For the readout function, the authors use JK with sum pooling.

One of the important achievements of GIN is the analysis of common pooling operations. As we discussed earlier, the sum pooling has the highest representative power. In comparison to it, mean pooling is only capable of learning the distribution of elements in the multiset. To see this let there be two multisets. The first multiset is non-empty, while the other has k-times elements from the first set. Then the averaging cancels the scale factor k out which results in the same embeddings for the pair of graphs. Finally, the max pooling operation can learn sets but not multisets because the operation considers only one element for all repetitions in the multiset. The order of the operations based on their representative power is the following: sum, mean, max.

Chapter 4
Methodology

In this chapter, the methods approaching the Job-Shop Scheduling problem employing Deep Reinforcement Learning and Graph Neural Networks are presented. Instead of being an improvement of some existing methods, almost all of them represent a concrete solution with a specific architecture, application, and evaluation pipeline that makes it difficult to compare them and identify the best one. At the same time, most of them lack the availability of source code, and the ones that have source code available usually employ a specific set of constraints that make it difficult to reuse them in different environments. Hence, in the scope of the thesis, we focus on a task of dynamic JSSP problem with the objective of tardiness minimization. The goal of the chapter is to introduce the reader to the methods studied in the scope thesis, as well as, describe conducted experiments and the evaluation procedure.

4.1 Review of the Literature

Almost all methods that are covered in the following section construct the solution by selecting the best priority dispatch rule for each job based on the current system state. All of them fall into the category of the reactive methods introduced in Section 1.7. The main distinction of the methods besides their application is the way how they define the MDP and the DRL method employed for searching for the optimal policy.

4.1.1 Deep Multi-Agent Reinforcement Learning

Deep Multi-Agent Reinforcement Learning (deep MARL) is the group of methods presented in the doctoral thesis [8] that approaches the dynamic JSSP problem with machine breakdowns. In particular, the work introduces two types of state encoding, i.e. Deep MARL Minimum Repetition (Deep MARL-MR) and Deep MARL Abstract State (Deep MARL-AS), and two methods

of reward shaping, i.e. reward shaped through decomposition and surrogate reward shaping. The model is trained using the multi-agent DDQN algorithm following centralized training and a decentralized execution scheme. In other words, each machine on the shop floor has a dedicated agent that is trained on its own decisions and the decisions of other agents. During the evaluation phase, the agents act independently without any communication between them.

To start the discussion about state encoding, it is important to note that the use of PDRs for the action space can guarantee at least some level of performance because they are designed to minimize the specific criterion. At the same time, PDRs are pretty restrictive because they commonly consider only one feature of the job. For example, the job with minimum slack time can have the most remaining processing time. These characteristics can be difficult for the model to learn based on the state representation. Deep MARL-MR directly encodes the criteria computed by the PDRs in the form of the matrix of size 5×5. The first four rows represent the features of jobs in the machine queue, i.e. operation processing time, remaining job processing time, slack time of the job, available time of succeeding machine, and the current wait time of the job in the machine queue. The available time of the machine is the amount of time the machine needs to process all jobs in the queue. At the same time, the first four job features are the criteria computed by SPT, LWKR, MS, and WINQ priority dispatch rules that represent the action space. If there are more than 4 jobs in the machine queue, then jobs for the state representation are selected using the aforementioned PDRs without repetition until the pool of jobs from the machine queue is exhausted. After that, the repetition of job features is allowed and the rest of the matrix is filled using the aforementioned procedure. The last row of the matrix represents the job that was selected for processing and is about to arrive on the machine. If the arriving job exists, then the row consists of the following features, i.e. operation processing time, remaining job processing time, slack time of the job, available time of the current machine, and the time till the job arrival. Otherwise, the row will contain only the available time of the current machine and other values will be zeros. The resulting matrix has features of both jobs in the machine queue and the job that is about to arrive at the machine that allows us to learn the direct mapping from the state to the job. Hence, the action space is reduced to the state space, in the way that the agent only selects jobs whose features are represented in the state.

In comparison to the Deep MARL-MR state encoding, the Deep MARL-AS encoding is represented by the vector of shop floor indicators. The representation allows us to use any priority dispatch rule as the action and the original work uses the set of 4 PDRs, i.e. SPT, CR, MS, and WINQ. The list of indicators is presented in Table 4.1.

Table 4.1 Deep MARL-AS Indicator List, where CV stands for Coefficient of Variation, (JMQ) stands for jobs in k-th machine queue, AM_k is the estimated time of processing all jobs in k-th machine queue, $\mathcal{J}^{\text{NOW}}$ is the set of jobs on the shopfloor at the moment, $\mathcal{J}^{\text{C}}$ is the set of all completed jobs till the moment, O_i^c the number of completed operations for job J_i, $\mathcal{P}$ is the multiset of processing times for all jobs from $\mathcal{J}^{\text{NOW}}$.

Indicator	Expression
1. In System job number	$\lvert\mathcal{J}\rvert$
2. Size of machine queue	$\lvert\mathcal{J}^k\rvert$
3. Number of arriving jobs at the machine	$\lvert AJ^k\rvert$
4. Min processing time (JMQ)	$\min_{J_i\in\mathcal{J}^k} p_{i,k}$
5. Mean processing time (JMQ)	$\overline{p}^k = \frac{1}{\lvert\mathcal{J}^k\rvert}\sum_{J_i\in\mathcal{J}^k} p_{i,k}$
6. Cumulative processing (JMQ)	$\sum_{J_i\in\mathcal{J}^k} p_{i,k}$
7. CV of processing time (JMQ)	$\text{std}(\{p_{i,k} : \forall J_i \in \mathcal{J}^k\})/\overline{p}^k$
8. Min remaining processing time (JMQ)	$\min_{J_i\in\mathcal{J}^k} WR_i$
9. Mean remaining processing time (JMQ)	$\overline{WR}^k = \frac{1}{\lvert\mathcal{J}^k\rvert}\sum_{J_i\in\mathcal{J}^k} WR_i$
10. Cumulative remaining processing time (JMQ)	$\sum_{J_i\in\mathcal{J}^k} WR_i$
11. CV of remaining processing time (JMQ)	$\text{std}(\{WR_i : \forall J_i \in \mathcal{J}^k\})/\overline{WR}^k$
12. Min time until due (JMQ)	$\min_{J_i\in\mathcal{J}^k} TTD_i$
13. Mean time until due (JMQ)	$\overline{TTD}^k = \frac{1}{\lvert\mathcal{J}^k\rvert}\sum_{J_i\in\mathcal{J}^k} TTD_i$
14. Cumulative time until due (JMQ)	$\sum_{J_i\in\mathcal{J}^k} TTD_i$
15. CV of time until due (JMQ)	$\text{std}(\{TTD_i : \forall J_i \in \mathcal{J}^k\})/\overline{TTD}^k$
16. Min slack time (JMQ)	$\min_{J_i\in\mathcal{J}^k} S_i$
17. Mean slack time (JMQ)	$\overline{S}^k = \frac{1}{\lvert\mathcal{J}^k\rvert}\sum_{J_i\in\mathcal{J}^k} S_i$
18. Cumulative slack time (JMQ)	$\sum_{J_i\in\mathcal{J}^k} S_i$
19. CV of slack time (JMQ)	$\text{std}(\{S_i : \forall J_i \in \mathcal{J}^k\})/\overline{S}^k$
20. Mean slack time of arriving jobs AJ^k (JMQ)	$\frac{1}{\lvert AJ^k\rvert}\sum_{J_i\in AJ^k} TTD_i$
21. Available time ratio w.r.t. all machines on the shop floor	$AM_k/\sum_{m_k\in\mathcal{M}} AM_k$
22. CV of cumulative processing time of all machines on the shop floor	$\text{std}(\mathcal{P})/\overline{\mathcal{P}}$
23. Average completion rate of jobs in the shop floor	$\sum_{J_i\in\mathcal{J}^{\text{NOW}}} O_i^c/\sum_{J_i\in\mathcal{J}^{\text{NOW}}} O_i$
24. Current tardy rate	$\lvert J_i : J_i \in \{\mathcal{J}^{\text{C}} \wedge T_i < 0\}\rvert/\lvert\mathcal{J}^{\text{C}}\rvert$
25. Expected tardy rate	$\lvert J_i : J_i \in \{\mathcal{J}^{\text{NOW}} \wedge S_i < 0\}\rvert/\lvert\mathcal{J}^{\text{NOW}}\rvert$

The reward the method aims to maximize must be directly related to the objective function of the JSSP. The thesis focuses on the tardiness minimization objective. One of the most natural reward functions for each active decision along the path of job $J_i \in \mathcal{J}$ through the shop floor is the following:

$$r_i = -T_i \tag{4.1}$$

where T_i is the tardiness of the job J_i. However, this reward function has several drawbacks. At first, the reward is available only after the job is completed, because the tardiness is realized at the completion time of the job. For off-policy methods like DQN, it is not a problem in episodic and continuing tasks, because the state-action-reward tuple can be stored in the replay buffer when the reward is available. However, for on-policy methods like PPO, it is a significant technical challenge to use it in the continuing task because the truncated trajectory can't be sampled without discarding some of the agent-environment interactions. The second drawback from the perspective of the MARL algorithm is that all agents receive the same reward although each decision along job path has a different impact on the tardiness of the job. Finally, the reward function is unbounded resulting in a high variance of the expected return. One of the ways to mitigate the last issue is to normalize reward by a constant factor c and clip it to the range $[-1, 0]$, i.e.:

$$r_i = \text{clip}(-\frac{T_i}{c}, -1, 0) \tag{4.2}$$

Reward shaping through decomposition proposed by [8] is a method that weights the reward of each agent by the impact of the decision on the tardiness of the job. The contribution of each machine is incorporated by considering the wait time in the machine queue and the slack time of the job in the queue. The decomposed reward for the tardy job is computed in the following 3 steps:

- **Reconstruction of queue time**: Let $q_{i,k} = r_{i,k} - a_{i,k}$ be the queue time the job waits in the machine queue. The reconstructed queue time for job J_m is computed as follows:

$$RQ_i = (1-\alpha)q_{i,k} + \alpha q'_{i+1,k}, \text{ where } q'_{i+1,k} = \begin{cases} q_{i+1,k} & \text{if } i+1 < O_m \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

  where $\alpha = 0.2$ is the hyperparameter. Agent decisions must follow two objectives, i.e. the minimization of wait time for tardy jobs in the machine queue and at the same time the minimization of the overall system congestion. By shifting back queue time from the succeeding machine, the agent is encouraged to select the job that satisfies both objectives.

- **Criticality of the job**: At the same time, the agent is punished for detaining jobs with short slack time. To achieve this each decision is weighted by the criticality factor, which is defined as follows:

$$CR_i = 1 - \frac{S_i}{|S_i| + \gamma} \tag{4.4}$$

  where S_i is the slack time of the job J_i and $\gamma = 128$ is the hyperparameter representing the sensivity to slack time.

- **Reward shaping:** The final reward for the tardy job is computed as follows:

$$r_i = \text{clip}(-(CR_i + RQ_i/\psi)^2, -1, 0) \tag{4.5}$$

  where $\psi = 128$ is the hyperparameter that normalizes RQ_i. Finally, the sum of $CR_i + RQ_i/\psi$ is squared to penalize the agent for excessively long wait times in the machine queue.

The received reward function is capable of measuring the impact of the agent's decision on the tardiness of the job, but it is still available only after the job is completed. The surrogate reward shaping makes it possible to compute the reward after the job is produced on the machine by stepping away from the global objective of tardiness. The minimization of tardiness is equivalent to the preservation of the slack time, i.e. if the job possessing more slack time is less likely to be tardy. At the same time, the preservation of the slack time is equivalent to the minimization of the wait time in the machine queue. In the loaded system, it is not always possible to preserve the slack time of all jobs, because queueing is inevitable. Hence, it is more reasonable to focus on jobs that are more likely to be tardy. To select such jobs, we can use the criticality factor defined in the previous step. Now, when the importance of the selected jobs is defined, we can focus on the machine decision. From the machine's perspective, the selection of the job has two side effects. At first, the machine ends the queue time of the selected job by extending the queue time of other jobs in the machine queue. At the same time, the selected job is exposed to the queueing at the next machine. The change in the queue time caused by the machine decision is derived by considering these two side effects. Let J_s be the job selected by the k-th machine decision and $\hat{J}$ be the set of the job left in the machine queue, NM_i be the next machine for the job J_i, AM_k be the available time of the m-th machine, and ξ be the hyperparameter. The reward function is defined as follows:

- **Queue time change for the selected job:**

$$\Delta S_1 = CR_s \times \frac{\sum_{J_i \in \hat{J}} t_{i,k}}{\left|\hat{J}\right|} - \xi \times AM_{NM_s} \tag{4.6}$$

- **Queue time change for the not selected jobs:**

$$\Delta S_2 = \frac{\sum_{J_i \in J^k} CR_i}{|J^k|} \times t_{s,k} - \xi \times \frac{\sum_{J_i \in \hat{J}} AM_{NM_i}}{\left|\hat{J}\right|} \tag{4.7}$$

- **Reward:**

$$r_t = \Delta S_1 - \Delta S_2 \tag{4.8}$$

- **Normalization and clipping:**

$$r'_t = \text{clip}(-\frac{r_t}{c}, -1, 1) \tag{4.9}$$

Hyperparameters ξ and c are found empirically for each problem configuration. In the context of the thesis, the configuration of the simulator similar to Deep MARL is used for all experiments, hence, we reuse the value from the original work $\psi = 0.2$ and $c = 30$.

4.1.2 Graph Neural Networks and other methods

A subset of methods employing Graph Neural Networks studied in the context of the thesis is presented in Table 4.2. Most of them solve the dynamic variant of the JSSP problem with the task of makespan minimization. The reward is defined as the difference between the quality measure $q(\cdot)$ of partial solutions at states s_t and s_{t+1}, i.e.

$$r_t = q(s_t) - q(s_{t+1})$$

The most common quality measure is the maximum lower bound of the operation completion time, $q(s_t) = \max_{o_{i,j} \in \mathcal{O}_{s_t}} C_{LB}(o_{i,j})$, where $\mathcal{O}_{s_t}$ is the set of all operation for jobs presented at the shop floor in the system state s_t. Let J_j be the job, r_i^j be the release time of operation $o_{i,j}$, the lower bound of the completion time is defined as follows [86]:

$$C_{\text{LB}}(o_{i,j}) = \begin{cases} r_i^j & \text{if } o_{ij} \text{ is completed} \\ C_{\text{LB}}(o_{i-1,j}) + p_i^j & \text{otherwise} \end{cases}$$

To show the idea behind the reward, let's compute the undiscounted return for a single episode.

$$\begin{aligned} \sum_{t=0}^{T} r_t &= q(s_0) - q(s_1) + q(s_1) - q(s_2) + \ldots + q(s_{T-1}) - q(s_T) = q(s_0) - q(s_T) \\ &= q(s_0) - q(s_{\text{T}}) = q(s_0) - C_{\max} \end{aligned} \tag{4.10}$$

Table 4.2 Overview of RL methods used to solve the JSSP problem. By direct action, we mean that the agent selects the operation to be processed from all jobs presented in the machine queue.

Title	Task	Action	Reward	GNN	RL
1. An End-to-end Hierarchical Reinforcement Learning Framework for Large-scale Dynamic Flexible Job-shop Scheduling Problem [46]	DFJSP	Direct	Makespan	GIN	PPO
2. Combining Reinforcement Learning Algorithms with Graph Neural Networks to Solve Dynamic Job Shop Scheduling Problems [87]	DJSP	6 PDRs	Custom	GCN	DQN
3. Deep Reinforcement Learning for dynamic flexible job shop scheduling problems considering variable processing times [88]	DFJSP	4 PDRs	Makespan	-	PPO
4. Dynamic Job-Shop Scheduling Problems Using Graph Neural Networks and Deep Reinforcement Learning [89]	DJSP	Direct	Makespan	Custom (MLP)	PPO
5. Dynamic job-shop scheduling using graph reinforcement learning with auxiliary strategy [90]	DJSP	8 PDRs	Makespan	Graph Trans-former	Truly PPO [91]
6. Flexible Job-Shop Scheduling via Graph Neural Network and Deep Reinforcement Learning [92]	FJSP	Direct	Makespan	GATv2	PPO
7. Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning [86]	JSP	Direct	Number of the waiting jobs	GIN	PPO

where $q(s_0)$ is a constant. From the last equation, we can observe that the maximization of return is equivalent to the minimization of the makespan.

While the minimization of makespan has its rationale, in the context of the dynamic JSSP problem, it is not the best metric as it was discussed in Section 1.3. At the same time, it is not clear how the quality measure $q(\cdot)$ is updated when the new job arrives on the shop floor because the arrival leads to the increase $C_{LB}(\cdot)$, which results in the reward independent of the machine decision. While the possible solution can be to ignore newly arrived jobs during reward computation, we couldn't find any discussion on the subject in the literature. Alternatively, the work [87] proposes the reward function for just-in-time scheduling. However, the reward is obtained by comparing the current solution and the best-achieved one, which makes it difficult to compute the reward in the continuing task.

From the perspective of the Graph Neural Networks, we can observe a high diversity of applied methods. Most of them employ well-known graph embedder architectures discussed in Chapter 3. One of the papers [90] proposes the transformer architecture for the JSSP problem. At the same time, they show that their method slightly outperforms the GATv2, hence, the latter is used during the evaluation. From the perspective of the MDP, the state is represented by the disjunctive graph or its modification which is discussed in Section 1.5. The action space is often modeled as the distribution over operation nodes representing jobs in the machine queue. Alternatively, some papers employ a set of PDRs that ensures the lower boundary on the performance of the model. Finally, from the perspective of the RL methods, the majority of works employ the PPO algorithm or its modification.

4.2 Experimental evaluation of Deep MARL

To experimentally evaluate the performance of the Deep MARL method a series of studies is conducted to identify, whether the adjustment in the architecture of the model can lead to a significant performance improvement. There are two main reasons for a such decision. Firstly, the performance improvement proposed by the Deep MARL is not that significant in comparison to regular PDRs. To be more specific, the Deep MARL-MR is only 5 percent better on average than the MS rule that it uses to select the job based on the results from the original work. At the same time, due to the implementation of methods and the simulator from scratch, we don't possess any knowledge about what works and what doesn't. Hence, the representative set of models is proposed after the initial set of experiments is performed.

The original work trains the DQN agent for 100000 seconds which is equivalent to the completion of 3100 jobs on average. The first 10000 seconds are dedicated to the warm-up process, in which the agent explores the environment with the FIFO rule. The training starts after the warm-up and the optimization step is performed every 5 seconds yielding a total of 19000 steps. There are two issues with the original training procedure that we address. At first, the interval of 5 seconds is too small to generate enough new interactions

for the agent to learn from, which from our perspective results in an excessively large number of optimization steps and, hence, higher computational costs. Secondly, the agent learns from the interactions with one environment, which may hurt the convergence speed of the model due to correlations between records in the replay buffer [93]. We propose to correct the first issue by performing the optimization step when 32 new records are stored in the replay buffer. The second issue is addressed by training the agent on 5 different environments running in parallel. Finally, instead of interacting with the environment in long sequences, the agents learn from 150 simulations of 150 jobs each. By performing these corrections the pool of training data is increased 7 times and the training time is increased only by a factor of 2.

To train the Deep MARL agent with the PPO we train the agent when 3 episodes of agent-environment interactions are collected for a total of 300 episodes. The agent is trained for 10 epochs with a batch of size 392. The number of simulations is increased in comparison to DQN because the PPO method is on-policy and must be trained on several episodes. At the same time, the computational complexity of the PPO doesn't change significantly, because, in comparison to DQN, the number of optimization steps is roughly the same.

4.3 Experimental evaluation of Graph State Encoding

The experimental evaluation of graph state encoding is performed by studying a set of candidates proposed by us following the approaches used in the literature. The training procedure for DQN agents is the same as for Deep MARL. Agents trained with PPO are trained on 500 simulations, but the training stops when the average entropy of decisions per the last 5 episodes is less than 0.25. The training starts with a high value of entropy regularization coefficient, i.e. 0.5, and decays by a factor of 0.995 after every optimization step. The adjustment is made due to the variability of entropy development for different architectures, which may lead to the overfitting of models that were converging faster than others. For GNN the state of the environment is represented by the disjunctive graph. As it was discussed in Section 1.5, the disjunctive graph contains a complete graph of operations representing jobs in the machine queue, i.e. the number of edges is $O(|\mathcal{J}^k|^2)$, which results in the high computational complexity of the system with a large number of jobs in machine queue. The problem becomes even more severe with RL methods because to train the model around several thousands of graphs must be stored in the memory. To address the issue, we propose to prune the disjunctive graph by selecting only jobs that are in the machine queue and the jobs that are next to arrive at the machine, which is inspired by the Deep MARL-MR state encoding. Additionally, we consider the modification of the graph from

[92] where authors propose to add a node representing the machine to the graph. Then the machine node is connected with undirected edges to operations representing jobs in the machine queue and jobs that were processed by the machine. The modification results in $\mathcal{O}(|\mathcal{J}|^k)$ edges to represent the connections between all operations for jobs in the machine queue.

For the evaluation, the architecture of the GNN is proposed based on the approaches from the literature. At first, node features are passed through a single-layer MLP with the LeakyRelu activation function. The obtained node representations are used as input to 3-layer Jumping Knowledge GNN, i.e. hidden representations of nodes at each graph layer are concatenated and passed through the MLP with the LeakyRelu activation function. Finally, node representations are aggregated following the requirements of specific tasks, which depend on the training algorithm and the action space. For the graph readout, the mean pooling is chosen. The output of the GNN is passed through the MLP with the LeakyRelu activation function. For all layers, the size of the hidden dimension is set to 128.

The studied set of candidates consists of all combinations of the following components:

- **Action space:** The action space can be modeled directly (D) as the distribution over operation nodes representing jobs in the machine queue. To construct the distribution only latent representations of the aforementioned nodes are considered. Alternatively, the action space is encoded indirectly (I) by the set of PDRs. In this case, the graph latent representation is obtained by the graph readout of operations representing jobs in the machine queue. From the obtained latent representation, the distribution of actions is constructed by decoding the representation with the MLP and softmax activation function.
- **Graph Layer:** Following the literature two most popular GNN architectures are either GATv2 or GIN, hence, they are selected for evaluation. For GATv2 the number of heads is set to 2.
- **Graph Encoding:** Two graph encodings are considered, i.e. the disjunctive graph (D) and the graph constructed with machine nodes (M).
- **Node Features:** Different papers propose different sets of operation features added to the node. In the scope of the thesis, two sets are considered. The first one is based on the paper [46] and we call it *hierarchical* (H). In the original work, each node contains the lower bound of operation completion time $C_{LB}(o_{i,j})$ and the status of the operation, i.e. a binary flag indicating whether the operation was completed or not. To incorporate the information about the job priority, we propose to add the estimated value of slack time of the job J_i derived from the $C_{LB}(o_{i,j})$, i.e.

$$S(o_{i,j}) = D_i - C_{LB}(o_{i,j})$$

where D_i is due time. When the job is completed, the estimated value of slack time is equal to the actual slack time of the job at the moment of the job production on the machine, otherwise it is the upper bound of the future slack time. The second set of features is proposed by us and we call it *custom* (C). It consists of three feature groups. The first group contains operation features, i.e. the lower bound of operation completion time $C_{LB}(o_{i,j})$, the operation processing time $p_{i,j}$, the estimated value of slack time $S(o_{i,j})$, criticality factor derived from the estimated value of slack time, and the completion rate of the j-th job at the i operation $cr_{i,j}$. The second group consists of binary indicators, i.e. whether the operation has arrived at the machine and whether the machine has started and finished the operation. The last group contains indicators to improve the observability of the model, i.e. the utilization rate of the machine, the wait time of the job in the machine queue, the expected tardy rate from Deep MARL-AS, the number of jobs arriving at the machine, the available time of the machine, the available time of the next machine, and the indicator of the next machine breakdown. To decorrelate the features from the last group from the number of jobs in the machine queue, the features are normalized by the number of jobs in the machine queue and they are added only to operations representing jobs in the machine queue. Finally, all time-related features are scaled down by the constant factor $c = 128$.

- **RL method:** Models are trained with both DDQN and PPO.

There are a total of 32 models. To refer to the candidate, the following notation is used: *Action Space - Graph Layer - Graph Encoding - Node Features - Training Algorithm*, e.g. I-GATv2-C-H-DDQN refers to the candidate with the indirect action space, GATv2 graph layer, disjunctive graph encoding, hierarchical node features, and DDQN training algorithm. For each training method and each action space, we consider only two candidates that achieved the best performance on the validation set.

4.4 Evaluation Procedure

For experiments, a new simulator was developed and it is available at [94]. The simulator supports the DJSSP and homogeneous DFJSSP problem with machine breakdowns and random arrival of the jobs. From the job perspective, the simulator is capable of generating jobs following the description presented in Section 1.10. At the same time, it can be applied to other JSSP problems with slight modifications because of the codebase quality from the perspective of software development. The simulator architecture is inspired by the [8] simulator, but we significantly refactored the codebase to support the experimental evaluation of different methods. The evaluation of the agent is performed by

simulating the production of 150 jobs under the same configuration of the dynamic JSSP problem and different random seeds. For each run, the cumulative tardiness of the first 100 produced jobs is measured. The cumulative tardiness of remaining jobs is not considered in the evaluation due to the high number of passive decisions caused by no arrival of new jobs. At the same time, the performance is not measured by computing the cumulative tardiness at the specific moment of the simulation like in [8], because the cumulative measure is computed over completed jobs that depends on the behavior of the agent. Finally, a lot of passive decisions may happen at the beginning of the simulation, when machine queues are filling up. Hence, each machine starts with 3 jobs in the queue.

The baseline rule for all experiments is the FIFO priority dispatch rule which represents the most passive decision-making strategy. To measure relative performance, the normalized cumulative tardiness is selected as the metric that is computed in the following way for each run:

$$\text{Normalized Performance} = \frac{\sum_i^{|\mathcal{J}|}(T_i^{\text{FIFO}} - T_i^{\text{Agent}})}{\sum_i^{|\mathcal{J}|} T_i^{\text{FIFO}}} \tag{4.11}$$

where $\mathcal{J}$ is the set of completed jobs.

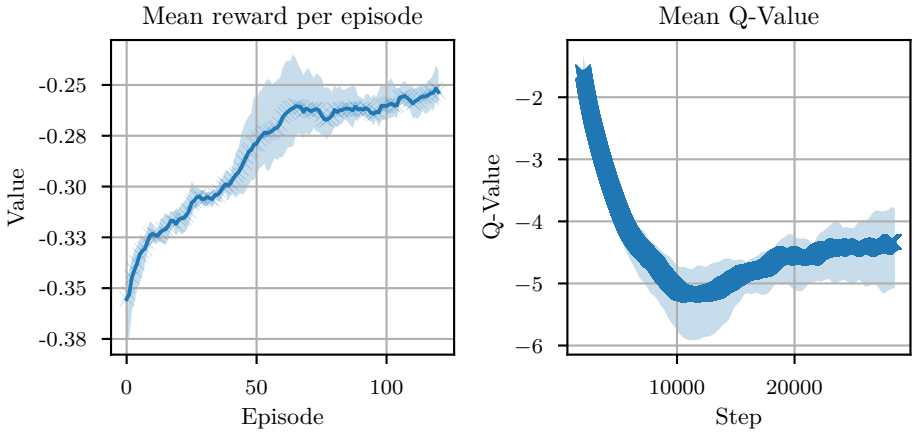
To minimize the effect of randomness, every model is trained 3 times with different random seeds. Then all agents are evaluated on 100 randomly generated tasks to produce 150 jobs under the same configuration of dynamic JSSP problem. Note, that evaluation instances are randomly generated, but they are the same for all agents. The evaluation of agents on static instances is not considered because they lack the information about due dates of the job. The quality of trained agents is then compared following the methodology presented in [95]. The first step of the procedure ranks all methods by the normalized performance for every run. Then the Friedman test is performed to determine whether there is a significant global difference between the models. Once the null hypothesis is rejected, a series of post-hoc tests is performed to determine the concrete pairwise differences between the models. In the context of the

Table 4.3 Comparison of 7 dispatching rules with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 1.7 \cdot 10^{-10}$.

Rule	Avg. Rank
CR+SPT	1.26
SPT	2.39
LWKR	3.29
CR	4.03
MS	4.97
WINQ	5.18
LIFO	6.88

Hypothesis	p
CR+SPT vs. SPT	0.001
SPT vs. LWRK	0.009
LWKR vs. CR	0.030
CR vs. MS	0.008
MS vs. WINQ	0.491
WINQ vs. LIFO	$2.36 \cdot 10^{-7}$

Figure 4.1 The development of the average reward per episode and average Q-values during the training of the DQN agent.



thesis, we choose Shaffer's multiple hypothesis testing [96]. The Shaffer's procedure is performed with the significance level of $\alpha = 0.05$. The example of the evaluation is presented in Table 4.3. We can observe that the CR+SPT rule is the best rule for all candidates. At the same time, there is not enough evidence to claim the difference between the MS and WINQ rules. It must be noted that the implementation of the procedure from [95] is used and we found out that doesn't rank correctly negative measures. Hence, normalized performances are standardized and shifted to the positive domain. The correction doesn't affect the results of any of the tests, because all of them are based on relative rankings of models over tasks that are not affected by the transformation.

For RL methods, it is important to define the behavior of agents during the evaluation phase. For instance, the DQN was evaluated on Atari games with .05-greedy behavior and random starts to minimize the effect of overfitting [51]. In comparison, Deep MARL agents act greedily during the evaluation phase [8]. At the same time, agents trained with the PPO method learn the stochastic policy that for some problems, e.g. poker is the optimal policy. In the context of the thesis, DQN agents are evaluated with greedy behavior, while PPO agents are evaluated stochastically.

Finally, it is important to indicate, if the agent learns from the interactions. For agents trained with DQN, there are two ways to analyze the learning process. The first one is to observe the trend of cumulative undiscounted returns for each run. Reinforcement learning maximizes the cumulative return, hence, a positive trend in return per episode should be observed. The second one is the development of the q-function values during training. As presented in [51] the absolute value of q-function values should increase during the training process. The development of the average reward per episode and the q-function values

Figure 4.2 The development of the average reward per episode and average Q-values during the training of the PPO agent.

of the Deep MARL-MR model are presented in Figure 4.1. Both indicators tell that the model is learning.

For agents trained with PPO, the development of average entropy per episode during the training is measured [72]. In general, the slow decay in the absolute value of the entropy should be observed, which indicates the shift of the model from active exploration to exploitation. Otherwise, the indicators of no learning process are either the maximum possible entropy, which means that the agent has learned random policy, or the steep decay of the entropy to zero, which indicates the agent lacks exploration of the environment. To achieve the aforementioned behavior, we empirically select the hyperparameters presented that can be found in [94]. The development of the entropy and the average return per episode for Deep MARL-MR are presented in Figure 4.2. Overall, the expected development of the entropy is observed, but at the same time, we can observe a high sensitivity of the method to the data it was trained on.

Chapter 5

Evaluation

In this chapter, we conduct the experimental evaluation of methods introduced in the previous chapter. The chapter starts with a description of the simulation environment and the evaluation of priority dispatch rules introduced in the previous sections. Then it proceeds with the description of the series of experiments conducted to evaluate the performance of Deep MARL methods and Graph Neural Networks to gain insights into the performance of the studied methods. Finally, the chapter concludes with the final evaluation of the methods obtained by the incorporation of obtained insights.

5.1 Simulation Parameters

DJSSP simulation parameters for all experiments are presented in Table 5.1. In general, we follow parameters from the Deep MARL [8], but we additionally consider the variability in job length and the noise in the processing time.

Table 5.1 Simulation parameters

Number of machines	10
Processing time p	Uniform$[1, 50]$
Processing time noise	$\mathcal{N}[0, 10^2]$
Tightness	Uniform$[1, 3]$
Utilization rate	0.8
Job length	$\{5 \dots 10\}$
Machine breakdown arrival	Exponential$[0.0002]$
Breakdown duration	Uniform$[200, 300]$

5.2 Experimental Evaluation of Priority Dispatch Rules

PDRs discussed in Section 1.7 are evaluated on the same set of tasks as the DRL agents. To show the relative strength of PDRs on the configuration of the DJSSP problem, we present the average ranking of the rules in Table 5.2. Overall, we can observe that composite rules outperform simple ones.

Table 5.2 Average ranking of 33 PDRs studied in the scope of the thesis.

Rule	Avg. Rank
CR+SPT	4.45
SPW +SPT	4.45
MOD	7.35
2PT+WINQ+NPT	7.81
GP2	8.19
LWKR+SPT	8.29
2PT + LWKR	8.29
COVERT	8.65
SPT	9.58
ATC	10.38
LWKR + MOD	10.82
PT + WINQ	11.38
AVPRO	12.27
GP1	13.61
PT + WINQ + S	14.05
LWKR	14.63
2PT + LWKR +S	15.96
MDD	17.00
SPW	17.88
CR	17.88
EDD	18.17
MS	20.76
WINQ	21.32
LRO	21.63
MON	22.03
NPT	26.59
LIFO	27.70
LWT	27.87
Random	27.99
SWT	28.13
MRO	31.45
LPT	32.04
MWKR	32.40

5.3 Rule Sets

In the scope of the chapter, the following rule sets are considered:

- **Deep MARL-MR rule set (MR)**: The set of rules used in the Deep MARL-MR, i.e. SPT, CR, LWKR, WINQ.
- **Deep MARL-AS rule set (AS)**: The set of rules used in the Deep MARL-AS, i.e. SPT, CR, MS, WINQ.
- **Rule set extended with due dates rules (E)**: The extended set of rules containing all rules from Deep MARL-MR and Deep MARL-AS, as well as, two additional rules involving due dates of the job, i.e. SPT, CR, MS, WINQ, EDD, LWRK, ATC.

- **Rule set extended with rules showing good performance (S)** - The extended set of rules containing all rules from Deep MARL-MR and Deep MARL-AS, as well as, rules that have strong performance on the current configuration of the problem, i.e. MS, SPT, LWKR, WINQ, CR+SPT, SPW+SPT, MOD, COVERT, ATC, 2PT+WINQ+NPT.

5.4 Experimental Evaluation of Deep MARL

For the study of Deep Multi-Agent Reinforcement Learning, the Deep MARL-MR state encoding and reward shaped through the decomposition are selected. The reader can find exact hyperparameters used for the training in [94].

At first, the impact of the architecture on the performance of the model is evaluated. The original work uses a small feedforward neural network with 6 hidden layers of sizes (64, 48, 48, 36, 24, 12) and the Tanh activation function. We note that the number of parameters can be too small for the complexity of the problem. Hence, we compare the performance of the model with two larger models with 3 hidden layers of sizes (256, 256, 256) and Relu/Tanh activation functions. Candidates are named by the activation function and the baseline model is Deep MARL-MR. The results of the evaluation procedure are presented in Table 5.3. At first, we can observe that none of the candidates nor the baseline outperforms the priority dispatch rule it uses, although, all models have achieved a similar high return per episode. The main reason behind the poor performance of the model seems to be the difference between the DJSSP problem configurations used in the original work and the thesis or possible differences in the simulation environments.

Secondly, the impact of the optimizer on the performance of the model is evaluated. In the previous experiment, the model was trained with the Adam optimizer and the learning rate of 0.001. We evaluate a series of optimizers, i.e. SGD with momentum, Adam [97], AdamW [98], AdaMax [97], RMSProp, RAdam [99], and non-stationary Adam. Non-stationary Adam [100] is the configuration of the Adam, that sets the same value for the coefficients used for

Table 5.3 Comparison of 4 dispatching rules and 3 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0.0$.

Rule	Avg. Rank
SPT	1.39
LWRK	2.21
MS	3.64
WINQ	3.89
Baseline	5.48
RELU	5.68
Tanh	5.70

Hypothesis	p
WINQ vs. Baseline	$1.38 \cdot 10^{-18}$
Baseline vs. Tanh	0.64
Baseline vs. RELU	0.64
RELU vs. Tanh	0.91

Table 5.4 Comparison of 4 dispatching rules and 7 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0.0$. Adam (NS) stands for non-stationary Adam.

Rule	Avg. Rank
SPT	1.76
LWRK	2.73
MS	4.72
WINQ	5.10
SGD	6.48
AdamW	6.54
Adam (NS)	6.85
AdaMax	7.62
Adam	7.64
RAdam	7.69
RMSProp	8.89

Hypothesis	p
WINQ vs. SGD	$7.25 \cdot 10^{-7}$
SGD vs. AdamW	3.108
AdamW vs. Adam (NS)	1.149
Adam (NS) vs. AdaMax	0.037
AdaMax vs. Adam	3.108
Adam vs. RAdam	3.108
RAdam vs. RMSProp	$1.78 \cdot 10^{-4}$
Adam (NS) vs. SGD	1.149
Adam (NS) vs. Adam	0.033
SGD vs Adam	$3.25 \cdot 10^{-4}$
Adam W vs. Adam	$7.03 \cdot 10^{-4}$

Table 5.5 Comparison of 4 dispatching rules and 2 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0.0$.

Rule	Avg. Rank
SPT	1.41
LWRK	2.16
MS	3.49
WINQ	3.78
Xavier	4.96
Orthogonal	5.20

Hypothesis	p
WINQ vs. Xavier	$3.77 \cdot 10^{-14}$
Xavier vs. Orthogonal	0.12
WINQ vs. Orthogonal	$5.79 \cdot 10^{-20}$

computing running averages of gradient (β_1, β_2) and the small value of weight decay. The authors claim that it prevents the policy forgetting of the PPO on some tasks. The results of the evaluation are presented in Table 5.4. Similarly, to the previous experiment, the model performs poorly in comparison to the priority dispatch rule it uses. Yet, we can observe that non-stationary Adam, SGD, and AdamW optimizers found better models than Adam, which is confirmed by the rejection of hypotheses Adam vs. Adam (NS), Adam vs. SGD, and Adam vs. AdamW.

In the third experiment, the impact of the initialization on the performance of the model is evaluated. The original work initializes the weights of the model with the Xavier initialization [101]. Additionally, the orthogonal initialization is considered, which was shown to be beneficial for the training of the deep neural networks [102]. The results of the evaluation are presented in Table 5.5. No significant difference between the two initializations is observed.

Then, the impact of the Deep MARL-AS state encoding on the performance of the model is evaluated. In comparison to the Deep MARL-MR that encodes

Table 5.6 Comparison of 4 dispatching rules and 3 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 2.37 \cdot 10^{-10}$. AS stands for Deep MARL-AS encoding, AS (E) stands for Deep MARL-AS with the extended rule set, and MR stands for Deep MARL-MR state encoding.

Rule	Avg. Rank
SPT	2.17
AS (E)	2.64
AS	2.71
LWRK	3.42
MS	5.10
WINQ	5.45
MR	6.50

Hypothesis	p
SPT vs. AS (E)	0.025
AS (E) vs. AS	0.69
AS vs. LWKR	$2.52 \cdot 10^{-4}$
AS vs. MS	$2.04 \cdot 10^{-101}$

the actions directly into the state, the Deep MARL-AS learns to select the best PDRs based on system indicators. The original work uses the (AS) rule set. Additionally, a candidate with the (E) rule set is evaluated. The results of the evaluation are presented in Table 5.6. We can observe that the Deep MARL-AS encoding outperforms the Deep MARL-MR state encoding trained on the reward shaped through decomposition, which is the opposite of the results from the original work. At the same time, there is not enough evidence to claim the difference between rule sets.

In the next step, Rainbow improvements discussed in Section 2.3 are evaluated. The original work uses the DDQN method. Additionally, the set of 7 candidates is considered, i.e. DQN, Dueling DQN, Prioritized Experience Replay (Prioritized), Noisy Nets (Noisy), 3-step learning (3-step), all of the improvements (All), and all of the improvements without noisy nets, and 3-step learning (All-nn3). The results of the evaluation are presented in Table 5.7. Overall, we can observe that the Prioritized Experience Replay and 3-step learning led to the improvement in the performance of the model.

In the final step of the DQN experiment, the impact of the reward functions is evaluated. We consider the set of 4 rewards, i.e. clipped tardiness, reward shaped through decomposition, surrogate reward shaping, and the makespan. For the makespan reward function, the reward is only computed based on the jobs presented on the shop floor at the moment of the decision. For the global tardiness and makespan reward, we normalize the metric by the constant $c = 128$ and clip the normalized metric to the range $[-1, 0]$ and $[-1, 1]$ respectively. For reward functions from the Deep MARL, the same sets of hyperparameters as presented in the original work are used. Additionally, for each reward function, we consider DDQN and All-nn3 variants of the Rainbow improvements. The results of the evaluation are presented in Table 5.8.

We can observe that surrogate reward shaping and makespan reward functions led to significant performance improvement. To understand the main reason behind the improvement, we analyze decisions made by the agents dur-

Table 5.7 Comparison of 4 dispatching rules and 3 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0$.

Rule	Avg. Rank
SPT	1.56
LWRK	2.52
MS	4.48
WINQ	4.80
Prioritized	5.62
3-step	7.38
All-nn3	8.02
All	8.48
Noisy	8.48
DDQN	8.54
DQN	9.02
Dueling	9.10

Hypothesis	p
WINQ vs. Prioritized	0.082
WINQ vs. 3-step	$5.12 \cdot 10^{-7}$
Prioritized vs. 3-step	$5.63 \cdot 10^{-7}$
3-step vs. All-nn3	0.471
All-nn3 vs. All	0.827
All vs. Noisy	3.178
Noisy vs. DDQN	3.178
DDQN vs. DQN	0.786
DQN vs. Dueling	3.178

Table 5.8 Comparison of 4 dispatching rules and 8 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0$.

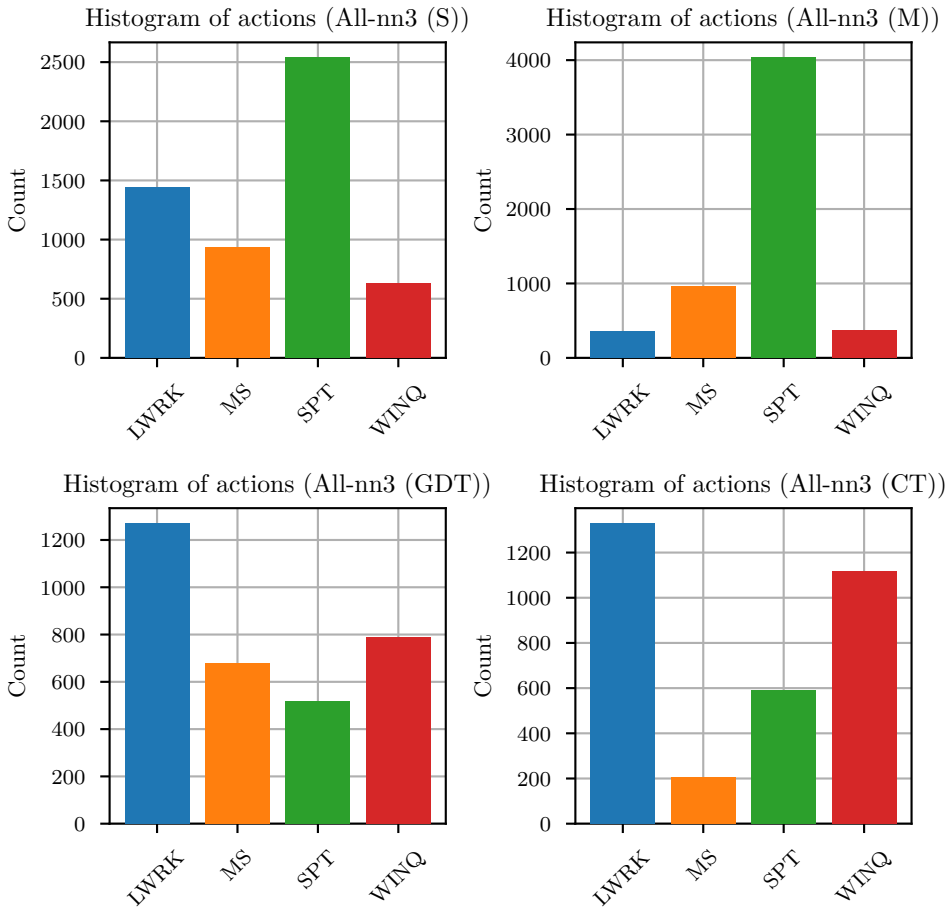
Rule	Avg. Rank
All-nn3 (S)	2.23
DDQN (S)	2.76
All-nn3 (M)	3.93
SPT	3.93
DDQN (M)	4.77
LWKR	5.43
MS	7.59
WINQ	8.02
DDQN (CT)	8.54
All-nn3 (CT)	10.06
DDQN (GDT)	10.28
All-nn3 (GDT)	10.47

Hypothesis	p
All-nn3 (S) vs. DDQN (S)	0.502
All-nn3 (S) vs. SPT	0.0029
DDQN (S) vs. All-nn3 (M)	$8.74 \cdot 10^{-4}$
All-nn3 (M) vs. SPT	1.323
All-nn3 (M) vs. DDQN (M)	0.043
SPT vs. DDQN (M)	0.043
DDQN (M) vs. LWKR	0.199
WINQ vs. DDQN (CT)	0.502
DDQN (CT) vs. All-nn3 (CT)	$4.33 \cdot 10^{-6}$
All-nn3 (CT) vs. DDQN (GDT)	1.323
DDQN (GDT) vs. All-nn3 (GDT)	1.323

ing the last 10000 steps of the training. At this moment of training the agent has 0.1-greedy behavior, hence, by analyzing the frequency of the actions we can understand the behavior of the agent. The results for the All-nn3 variant are presented in Figure 5.1. We can observe that the agents trained with the surrogate reward shaping and makespan reward functions are more likely to select the job following the SPT rule, which is known to be a good rule for makespan minimization [4]. While the agents trained with the clipped tardiness and reward shaped through decomposition are more likely to select the job following the LWKR and WINQ rules. At the same time, the All-nn3 variant resulted in better overall results, yet, there is not enough evidence to claim the difference between variants.

Finally, the PPO method is evaluated. The results of the evaluation are

Figure 5.1 Histogram of DQN decisions during the last 10000 seconds of the simulation. (S) stands for surrogate reward shaping. (M) stands for makespan reward. (CT) stands for clipped tardiness reward. (GDT) stands for reward shaped through decomposition.



presented in Table 5.9. While we formally reject the null hypothesis of the difference between the PPO and DDQN methods, we can not claim that the PPO method is better than the DDQN method due to the low performance of the obtained models.

Overall, after the series of 7 studies, we can observe that only some of the candidates possessed significant performance improvement which doesn't allow us to make strong statements about the strength of the adjustment. Yet, a series of the most prominent adjustments was found, which are selected for the final evaluation. These adjustments are the Deep MARL-AS state encoding, the surrogate reward shaping, the non-stationary Adam optimizer, the Prioritized Experience Replay, and the PPO method.

Table 5.9 Comparison of 4 dispatching rules and 2 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 1.89 \cdot 10^{-10}$.

Rule	Avg. Rank
SPT	1.41
LWKR	2.36
MS	3.49
WINQ	4.00
PPO	4.34
DDQN	5.22

Hypothesis	p
WINQ vs. PPO	0.052
PPO vs. DDQN	$2.94 \cdot 10^{-8}$

5.5 Experimental Evaluation of Graph Neural Networks

The experimental evaluation of Graph Neural Networks is performed on the set of models and the training procedure discussed in Section 4.3. All models are trained with the reward shaped through decomposition and the extended rule set. The results of the evaluation are presented in Table 5.10. The best models are the ones that were trained with DDQN and have direct action encoding and custom node features. These models have achieved their performance by selecting either the SPT or the ATC rule most of the time. The second group of models is the ones that were trained with PPO and have hierarchical node features. In comparison to the DDQN models, these models have learned more

Table 5.10 Comparison of 5 dispatching rules and 8 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0$. The abbreviation is read as follows, i.e. (Direct|Indirect action set) - (GATv2|GIN) - (Machine node|Complete graph encoding) - (Hierarchical|Custom node features) - (DQN|PPO).

Rule	Avg. Rank
SPT	3.01
D-GATv2-M-C-DQN (1)	3.01
D-GIN-C-C-DQN (2)	3.21
D-GATv2-M-H-PPO (3)	3.94
D-GATv2-C-H-PPO (4)	4.25
LWRK	4.76
MS	7.21
WINQ	7.76
I-GATv2-M-H-DQN (5)	9.70
I-GIN-C-C-PPO (6)	9.92
I-GIN-C-H-PPO (7)	10.38
I-GATv2-C-C-DQN (8)	11.69
Random	12.16

Hypothesis	p
SPT vs. (1)	2.86
(1) vs. (2)	2.86
(2) vs. (3)	2.03
(3) vs. (4)	2.86
(4) vs. LWRK	2.86
WINQ vs. (5)	0.01
(5) vs. (6)	2.86
(6) vs. (7)	2.86
(7) vs. (8)	0.31
(8) vs. Random	$2.30 \cdot 10^{-4}$

balanced policies, i.e. all rules have a high probability of being selected. Finally, all models with the indirect action encoding have performed poorly in comparison to the models with the direct action encoding. However, they have achieved better performance than the random policy. At the same time, all models with indirect action encoding have learned to obtain a high average reward per episode, which indicates that the problem may be with the reward function. Therefore, for the final evaluation, the same set of models is considered, but the reward function is changed to the surrogate reward shaping that has shown good performance in deep MARL experiments.

5.6 Final Evaluation

In the first step of the final evaluation, the set of 8 models is constructed based on the results of the previous studies. To describe the set we use the following notation: *State Encoding - Reward Function - Prioritized Replay Buffer - Architecture - Rule Set*. The state encoding is either Deep MARL-AS (AS) or Deep MARL-MR (MR). The reward function is either the surrogate reward shaping (S) or the reward shaped through decomposition (GDT). The model is either trained with a prioritized replay buffer (P) or not (N). The architecture is either from the original work (O) or the 3-layer MLP with RELU activation function and the hidden dimension of 256 (R). The evaluation results and models are presented in Table 5.11. Overall, it can be observed that the Deep MARL-MR with surrogate reward shaping trained with the original architecture and the base rule set is the only model that is capable of learning the policy better than the rules it uses. At the same time, the agent is unable to learn the policy that is better than CR+SPT which is one of the best rules for the current configuration of the DJSSP problem.

For the evaluation of PPO, the set of models is constructed similarly to

Table 5.11 Comparison of 5 dispatching rules and 8 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0$.

Rule	Avg. Rank
CR+SPT	2.38
MR-S-P-O-MR (1)	3.59
MR-S-N-O-MR (2)	4.21
SPT	4.81
AS-GDT-N-O-E (3)	5.24
AS-S-P-R-S (4)	6.49
AS-GDT-P-O-E (5)	6.73
LWRK	7.17
AS-S-P-O-E (6)	8.11
AS-S-N-O-E (7)	8.58
MS	10.09
WINQ	10.62

Hypothesis	p
CR+SPT vs. (1)	$4.56 \cdot 10^{-4}$
(1) vs. (2)	0.24
(2) vs. SPT	0.025
(1) vs. SPT	$4.56 \cdot 10^{-4}$
SPT vs. (3)	0.42
(3) vs. (4)	0.42
(4) vs. (5)	0.421
(5) vs. LWRK	0.421
LWRK vs. (6)	0.012
(6) vs. (7)	0.421
(7) vs. MS	$6.72 \cdot 10^{-6}$

Table 5.12 Comparison of 5 dispatching rules and 5 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0$.

Rule	Avg. Rank
CRSPT	1.60
SPT	3.40
AS-S-O-AS (1)	3.41
AS-GDT-O-E (2)	4.63
LWRK	4.97
AS-S-R-S (3)	5.68
MS	7.23
WINQ	7.71
MR-S-R-MR (4)	7.84
MR-S-O-MR (5)	8.54

Hypothesis	p
SPT vs. (1)	1.23
(1) vs. (2)	$7.47 \cdot 10^{-6}$
(2) vs. LWRK	0.51
LWRK vs. (3)	0.025
(3) vs. MS	$5.57 \cdot 10^{-9}$
WINQ vs. (4)	0.006
(4) vs. (5)	0.027

Table 5.13 Comparison of 5 dispatching rules and 8 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0$. The abbreviation is read as follows, i.e. (Direct|Indirect action set) - (GATv2|GIN) - (Machine node|Complete graph encoding) - (Hierarchical|Custom node features) - (DQN|PPO).

Rule	Avg. Rank
CRSPT	3.24
I-GATv2-M-C-DQN (1)	3.62
I-GINv2-M-C-DQN (2)	4.19
D-GATv2-C-H-DQN (3)	5.49
D-GATv2-C-H-PPO (4)	5.66
SPT	6.38
D-GATv2-M-C-DQN (5)	6.28
D-GATv2-M-C-PPO (6)	6.65
I-GIN-M-H-PPO (7)	8.47
LWRK	8.49
I-GIN-C-C-PPO (8)	9.11
MS	11.45
WINQ	11.96

Hypothesis	p
CRSPT vs. (1)	1.39
(1) vs. (2)	0.61
(1) vs. (3)	$7.14 \cdot 10^{-4}$
(3) vs. (4)	1.77
(4) vs. SPT	0.28
SPT vs. (5)	1.77
(5) vs. (6)	1.39
(6) vs. (7)	$2.22 \cdot 10^{-7}$
(7) vs. LWRK	1.77
LWRK vs. (8)	0.49
(8) vs. MS	$6.2 \cdot 10^{-11}$

the DQN. However, PPO doesn't use the experience replay buffer, hence, there are only 5 models considered in the evaluation. From the perspective of the entropy development variants AS-S-R-S and MR-S-R-S were terminated after around 170 episodes of simulation due to a fall in the entropy of the policy. The results of the evaluation are presented in Table 5.12. Overall, we can observe that with PPO method is unable to learn better policies than the DDQN method.

For graph evaluation, the same set of models is constructed, but the improvements from deep MARL are considered, i.e. the agent is trained with the surrogate reward shaping, DDQN agents use the prioritized replay buffer and for rule set, the (S) set of rules is used. The results are presented in Table 5.13. Similarly to Deep MARL-MR state encoding, the models with direct action

encoding perform significantly better than models encoding jobs indirectly.

Finally, the models from Deep MARL (MR-S-P-O-MR, MR-S-N-O-MR, AS-GDT-N-O-E) and GNN (I-GATv2-M-C-DQN, I-GINv2-M-C-DQN) that performed the best in comparison to CRSPT rule are evaluated on the new dataset of consisting of 200 tasks. The results are presented in Table 5.14. I-GATv2-M-C-DQN obtained statistically better results than any other trained model, yet, the performance of the model is not sufficient to outperform the CRSPT rule.

Table 5.14 Comparison of 10 dispatching rules and 5 candidates with the level of significance $\alpha = 0.05$. Friedman procedure rejects null-hypothesis with $p_{\text{Friedman}} = 0$.

Rule	Avg. Rank
CRSPT	3.52
I-GATv2-M-C-DQN (1)	4.01
MR-S-P-O-MR (2)	5.06
MOD	5.15
I-GINv2-M-C-DQN (3)	5.36
MR-S-N-O-MR (4)	5.45
2PT + WINQ + NPT	6.06
SPT	7.32
ATC	7.33
AS-GDT-N-O-E (5)	10.10
CR	10.20
MS	11.75
LWRK	11.31
WINQ	12.35

Hypothesis	p
CRSPT vs. (1)	0.43
(1) vs. (2)	$2.19 \cdot 10^{-4}$
(1) vs. (3)	$4.46 \cdot 10^{-7}$
(1) vs. (4)	$5.04 \cdot 10^{-8}$
(2) vs. (3)	2.675
(2) vs. (4)	0.82
(3) vs. (4)	2.67
(4) vs. (5)	$1.11 \cdot 10^{-8}$

5.7 Discussion

After evaluating a wide range of models and methods, we have observed that the best models are the ones that learn to select the job directly. We believe that the main reason behind the weak performance of the rule-based methods is the necessity of learning the result of rule application from the system state. This introduces additional complexity to the problem, which results in the poor performance of the models. An interesting direction of further research can be the direct indication of the job that will be selected by the rule in the system. By doing so almost all time-related information (e.g. operation processing time, remaining work, etc.) can be encoded in the form of binary indicators, which will make the input time-independent and help the model to generalize to other configurations of the DJSSP problem. From the perspective of the RL methods, we have observed that the DQN-based methods were yielding similar or sometimes even better results than the PPO. One of the possible reasons for the observed behavior of the PPO method can be some unknown implementation detail that we are not aware of [71]. Being concep-

tually simpler, we advise beginning with the DQN-based methods and then proceeding to the PPO, if the DQN-based methods are not sufficient. Therefore, further research can be conducted in the direction of improving DRL methods to solve the DJSSP problem. One of the interesting observations was the similarity between the makespan reward function and the surrogate reward shaping. Intuitively, blind minimization of tardiness may result in an overall increase in the flow time of the job through the system, which eventually will increase the total cumulative tardiness. Surrogate reward shaping seems to be the trade-off between the makespan and the tardiness that results in better overall outcomes for the DJSSP with job priorities. An interesting direction of further research can be the development of new reward functions that will consider several objectives at once. Finally, the configuration of the DJSSP problem used in the thesis may be too complicated for practical needs. Hence, further research by evaluating the methods under different configurations and evaluation of DRL methods on real-world data should be conducted.

Conclusion

The main goal stated at the beginning of the thesis is to perform a comparative study of the methods employing Deep Reinforcement Learning and Graph Neural Networks to solve the Dynamic Job Shop Scheduling Problem with the objective of the minimization of total cumulative tardiness. One work that focuses on our task is Deep MARL [8], which becomes the starting point of the simulation environment, training, and evaluation procedure. After the development of the simulator, we conduct a series of experiments to gain insights into the performance of the Deep MARL method. Then we propose a series of improvements to the training and evaluation procedure such as more efficient gathering of the training data, Rainbow, etc. We perform a series of experiments to evaluate the proposed improvements following the new evaluation procedure. A lot of methods in literature, that use GNN focus on the minimization of the makespan and don't consider job priorities represented by the due dates. To evaluate the performance of graph state representation on our task, we propose a set of 32 methods that cover the most important components of the studied methods. The evaluation of the set showed that one of the novel methods has shown a statistically significant improvement over the Deep MARL method. Yet, its performance is still slightly behind one of the PDRs studied in the scope of the thesis. Overall, we have evaluated a wide range of methods, proposed a various set of improvements, and provided a comprehensive evaluation of the methods employing DRL and GNNs to solve the DJSSP problem.

Bibliography

1. VIEIRA, Guilherme E.; HERRMANN, Jeffrey W.; LIN, Edward. Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods. *Journal of Scheduling*. 2003, vol. 6, no. 1, pp. 39–62. ISBN 1099-1425. Available from DOI: 10.1023/A:1022235519958.
2. ZHENG, Pai; WANG, Honghui; SANG, Zhiqian; ZHONG, Ray Y.; LIU, Yongkui; LIU, Chao; MUBAROK, Khamdi; YU, Shiqiang; XU, Xun. Smart manufacturing systems for Industry 4.0: Conceptual framework, scenarios, and future perspectives. *Frontiers of Mechanical Engineering*. 2018, vol. 13, no. 2, pp. 137–150. ISBN 2095-0241. Available from DOI: 10.1007/s11465-018-0499-5.
3. MUTH, J.F.; THOMPSON, G.L. *Industrial Scheduling*. Prentice-Hall, 1963. Industrial Scheduling. Available also from: https://books.google.cz/books?id=A5AgAAAAMAAJ.
4. Single-machine Sequencing. In: *Principles of Sequencing and Scheduling*. John Wiley & Sons, Ltd, 2018, chap. 2, pp. 11–37. ISBN 9781119262602. Available from DOI: https://doi.org/10.1002/9781119262602.ch2.
5. MOSHEIOV, Gur. Complexity analysis of job-shop scheduling with deteriorating jobs. *Discrete Applied Mathematics*. 2002, vol. 117, no. 1, pp. 195–209. ISBN 0166-218X. Available from DOI: https://doi.org/10.1016/S0166-218X(00)00385-1.
6. DAUZÈRE-PÉRÈS, Stéphane; DING, Junwen; SHEN, Liji; TAMSSAOUET, Karim. The flexible job shop scheduling problem: A review. *European Journal of Operational Research*. 2024, vol. 314, no. 2, pp. 409–432. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/j.ejor.2023.05.017.
7. OUELHADJ, Djamila; PETROVIC, Sanja. A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*. 2009, vol. 12, no. 4, pp. 417–431. ISBN 1099-1425. Available from DOI: 10.1007/s10951-008-0090-8.

8. LIU, Renke. Deep reinforcement learning-based dynamic scheduling. 2022.

9. YUGMA, Claude; DAUZÈRE-PÉRÈS, Stéphane; ARTIGUES, Christian; DERREUMAUX, Alexandre; SIBILLE, Olivier. A batching and scheduling algorithm for the diffusion area in semiconductor manufacturing. *International Journal of Production Research*. 2012, vol. 50, no. 8, pp. 2118–2132. ISBN 0020-7543. Available from DOI: 10.1080/00207543.2011.575090.

10. AYTUG, Haldun; LAWLEY, Mark A.; MCKAY, Kenneth; MOHAN, Shantha; UZSOY, Reha. Executing production schedules in the face of uncertainties: A review and some future directions. *European Journal of Operational Research*. 2005, vol. 161, no. 1, pp. 86–110. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/j.ejor.2003.08.027. IEPM: Focus on Scheduling.

11. URASEVIĆ, Marko; JAKOBOVIĆ, Domagoj. A survey of dispatching rules for the dynamic unrelated machines environment. *Expert Systems with Applications*. 2018, vol. 113, pp. 555–569. ISBN 0957-4174. Available from DOI: https://doi.org/10.1016/j.eswa.2018.06.053.

12. VERONIQUE SELS, Nele Gheysen; VANHOUCKE, Mario. A comparison of priority rules for the job shop scheduling problem under different flow time- and tardiness-related objective functions. *International Journal of Production Research*. 2012, vol. 50, no. 15, pp. 4255–4270. Available from DOI: 10.1080/00207543.2011.611539.

13. Simulation Models for the Dynamic Job Shop. In: *Principles of Sequencing and Scheduling*. 2018, pp. 427–452. ISBN 9781119262602. Available from DOI: https://doi.org/10.1002/9781119262602.ch15.

14. QUADRAS, Djonathan; FRAZZON, Enzo M.; MENDES, Lucio G.; PIRES, Matheus C.; RODRIGUEZ, Carlos M. T. Adaptive Simulation-Based Optimization for Production Scheduling: A Comparative Study. *IFAC-PapersOnLine*. 2022, vol. 55, no. 10, pp. 424–429. ISBN 2405-8963. Available from DOI: https://doi.org/10.1016/j.ifacol.2022.09.430.

15. PICKARDT, Christoph; BRANKE, Jürgen; HILDEBRANDT, Torsten; HEGER, Jens; SCHOLZ-REITER, Bernd. Generating dispatching rules for semiconductor manufacturing to minimize weighted tardiness. In: *Proceedings of the 2010 Winter Simulation Conference*. 2010, pp. 2504–2515. Available from DOI: 10.1109/WSC.2010.5678946.

16. NGUYEN, Su; MEI, Yi; ZHANG, Mengjie. Genetic programming for production scheduling: a survey with a unified framework. *Complex & Intelligent Systems*. 2017, vol. 3, no. 1, pp. 41–66. ISBN 2198-6053. Available from DOI: 10.1007/s40747-017-0036-x.

17. ZHANG, Fangfang; MEI, Yi; NGUYEN, Su; ZHANG, Mengjie. Survey on Genetic Programming and Machine Learning Techniques for Heuristic Design in Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation*. 2024, vol. 28, no. 1, pp. 147–167. Available from DOI: 10.1109/TEVC.2023.3255246.

18. LEE, YOUNG HOON; BHASKARAN, KUMAR; PINEDO, MICHAEL. A heuristic to minimize the total weighted tardiness with sequence-dependent setups. *IIE Transactions*. 1997, vol. 29, no. 1, pp. 45–52. ISBN 0740-817X. Available from DOI: 10.1080/07408179708966311.

19. NGUYEN, Su; ZHANG, Mengjie; JOHNSTON, Mark; TAN, Kay Chen. Genetic Programming for Job Shop Scheduling. In: *Evolutionary and Swarm Intelligence Algorithms*. Ed. by BANSAL, Jagdish Chand; SINGH, Pramod Kumar; PAL, Nikhil R. Cham: Springer International Publishing, 2019, pp. 143–167. ISBN 978-3-319-91341-4. Available from DOI: 10.1007/978-3-319-91341-4{_}8.

20. ZHANG, Fangfang; MEI, Yi; NGUYEN, Su; ZHANG, Mengjie. Evolving Scheduling Heuristics via Genetic Programming With Feature Selection in Dynamic Flexible Job-Shop Scheduling. *IEEE Transactions on Cybernetics*. 2021, vol. 51, no. 4, pp. 1797–1811. Available from DOI: 10.1109/TCYB.2020.3024849.

21. DAUZÈRE-PÉRÈS, Stéphane; PAULLI, Jan. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*. 1997, vol. 70, no. 0, pp. 281–306. ISBN 1572-9338. Available from DOI: 10.1023/A:1018930406487.

22. BRANDIMARTE, Paolo. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*. 1993, vol. 41, no. 3, pp. 157–183. ISBN 1572-9338. Available from DOI: 10.1007/BF02023073.

23. YSKA, Daniel; MEI, Yi; ZHANG, Mengjie. *Genetic Programming*. Genetic Programming Hyper-Heuristic with Cooperative Coevolution for Dynamic Flexible Job Shop Scheduling. Ed. by CASTELLI, Mauro; SEKANINA, Lukas; ZHANG, Mengjie; CAGNONI, Stefano; GARCÍA-SÁNCHEZ, Pablo. Cham: Springer International Publishing, 2018. ISBN 978-3-319-77553-1.

24. HERROELEN, Willy; LEUS, Roel. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research*. 2005, vol. 165, no. 2, pp. 289–306. ISBN 0377-2217. Available from DOI: https://doi.org/10.1016/j.ejor.2004.04.002.

25. SEVAUX, Marc; SÖRENSEN, Kenneth. A genetic algorithm for robust schedules in a just-in-time environment. *University of Antwerp, Faculty of Applied Economics, Working Papers*. 2002.

26. MINGUILLON, Fabio Echsler; LANZA, Gisela. Coupling of centralized and decentralized scheduling for robust production in agile production systems. *Procedia CIRP*. 2019, vol. 79, pp. 385–390. ISSN 2212-8271. Available from DOI: https://doi.org/10.1016/j.procir.2019.02.099. 12th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 18-20 July 2018, Gulf of Naples, Italy.

27. HAZIR, Öncü; HAOUARI, Mohamed; EREL, Erdal. Robust scheduling and robustness measures for the discrete time/cost trade-off problem. *European Journal of Operational Research*. 2010, vol. 207, no. 2, pp. 633–643. ISBN 0377-2217. Available from DOI: https://doi.org/10.1016/j.ejor.2010.05.046.

28. XIONG, Jian; XING, Li-ning; CHEN, Ying-wu. Robust scheduling for multi-objective flexible job-shop problems with random machine breakdowns. *International Journal of Production Economics*. 2013, vol. 141, no. 1, pp. 112–126. ISBN 0925-5273. Available from DOI: https://doi.org/10.1016/j.ijpe.2012.04.015.

29. SHELOKAR, Prakash; KULKARNI, Abhijit; JAYARAMAN, Valadi K.; SIARRY, Patrick. Metaheuristics in Process Engineering: A Historical Perspective. In: *Applications of Metaheuristics in Process Engineering*. Ed. by VALADI, Jayaraman; SIARRY, Patrick. Cham: Springer International Publishing, 2014, pp. 1–38. ISBN 978-3-319-06508-3. Available from DOI: 10.1007/978-3-319-06508-3{_}1.

30. DEB, K.; PRATAP, A.; AGARWAL, S.; MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*. 2002, vol. 6, no. 2, pp. 182–197. ISBN 1941-0026. Available from DOI: 10.1109/4235.996017.

31. JADAAN, Omar al; RAJAMANI, Lakishmi; RAO, C. Non-dominated ranked genetic algorithm for solving multi-objective optimization problems: NRGA. *Journal of Theoretical and Applied Information Technology*. 2008.

32. AHMADI, Ehsan; ZANDIEH, Mostafa; FARROKH, Mojtaba; EMAMI, Seyed Mohammad. A multi objective optimization approach for flexible job shop scheduling problem under random machine breakdown by evolutionary algorithms. *Computers & Operations Research*. 2016, vol. 73, pp. 56–66. ISBN 0305-0548. Available from DOI: https://doi.org/10.1016/j.cor.2016.03.009.

33. AL-HINAI, Nasr; ELMEKKAWY, T. Y. Robust and stable flexible job shop scheduling with random machine breakdowns using a hybrid genetic algorithm. *International Journal of Production Economics*. 2011, vol. 132, no. 2, pp. 279–291. ISBN 0925-5273. Available from DOI: https://doi.org/10.1016/j.ijpe.2011.04.020.

34. COLORNI, Alberto; DORIGO, Marco; MANIEZZO, Vittorio; TRUBIAN, Marco. Ant system for job-shop scheduling. *STATISTICS AND COMPUTER SCIENCE*. 1994, vol. 34.

35. S. G., Ponnambalam; N, Jawahar; BS, Girish. An Ant Colony Optimization Algorithm for Flexible Job Shop Scheduling Problem. In: 2010. ISBN 978-953-307-067-4. Available from DOI: 10.5772/9425.

36. WANG, Lei; CAI, Jingcao; LI, Ming; LIU, Zhihu. Flexible Job Shop Scheduling Problem Using an Improved Ant Colony Optimization. *Scientific Programming*. 2017, vol. 2017, pp. 1–11. Available from DOI: 10.1155/2017/9016303.

37. ZHANG, Rui; SONG, Shiji; WU, Cheng. Robust Scheduling of Hot Rolling Production by Local Search Enhanced Ant Colony Optimization Algorithm. *IEEE Transactions on Industrial Informatics*. 2019, vol. PP, pp. 1–1. Available from DOI: 10.1109/TII.2019.2944247.

38. LEON, V. Jorge; WU, S.; STORER, Robert. Robustness Measures and Robust Scheduling for Job Shops. *Iie Transactions*. 1994, vol. 26, pp. 32–43. Available from DOI: 10.1080/07408179408966626.

39. MEHTA, Sanjay V. Predictable scheduling of a single machine subject to breakdowns. *International Journal of Computer Integrated Manufacturing*. 1999, vol. 12, no. 1, pp. 15–38. ISBN 0951-192X. Available from DOI: 10.1080/095119299130443.

40. BEAN, James C.; BIRGE, John R.; MITTENTHAL, John; NOON, Charles E. Matchup Scheduling with Multiple Resources, Release Dates and Disruptions. *Operations Research*. 1991, vol. 39, no. 3, pp. 470–483. ISBN 0030-364X. Available from DOI: 10.1287/opre.39.3.470.

41. SONG, Lijun; XU, Zhipeng; WANG, Chengfu; SU, Jiafu. A New Decision Method of Flexible Job Shop Rescheduling Based on WOA-SVM. *Systems*. 2023, vol. 11, no. 2. ISBN 2079-8954. Available from DOI: 10.3390/systems11020059.

42. AL-BEHADILI, Mohanad; OUELHADJ, Djamila; JONES, Dylan. *Intelligent Systems Design and Applications*. Multi-objective Particle Swarm Optimisation for Robust Dynamic Scheduling in a Permutation Flow Shop. Ed. by MADUREIRA, Ana Maria; ABRAHAM, Ajith; GAMBOA, Dorabela; NOVAIS, Paulo. Cham: Springer International Publishing, 2017. ISBN 978-3-319-53480-0.

43. TAILLARD, E. Benchmarks for basic scheduling problems. *European Journal of Operational Research*. 1993, vol. 64, no. 2, pp. 278–285. ISBN 0377-2217. Available from DOI: https://doi.org/10.1016/0377-2217(93)90182-M.

44. DEMIRKOL, Ebru; MEHTA, Sanjay; UZSOY, Reha. Benchmarks for shop scheduling problems. *European Journal of Operational Research*. 1998, vol. 109, no. 1, pp. 137–141. ISBN 0377-2217. Available from DOI: https://doi.org/10.1016/S0377-2217(97)00019-2.

45. WEISE, Thomas. *jsspInstancesAndResults: Results, Data, and Instances of the Job Shop Scheduling Problem*. Hefei, Anhui, China: Institute of Applied Optimization, Hefei University, 2019–2020. Available also from: https://github.com/thomasWeise/jsspInstancesAndResults. A GitHub repository with the common benchmark instances for the Job Shop Scheduling Problem as well as results from the literature, both in form of CSV files as well as R program code to access them.

46. LEI, Kun; GUO, Peng; WANG, Yi; XIONG, Jianyu; ZHAO, Wenchao. An End-to-end Hierarchical Reinforcement Learning Framework for Large-scale Dynamic Flexible Job-shop Scheduling Problem. In: *2022 International Joint Conference on Neural Networks (IJCNN)*. 2022, pp. 1–8. Available from DOI: 10.1109/IJCNN55064.2022.9892005.

47. RANGSARITRATSAMEE, Ruedee; FERRELL, William G.; KURZ, Mary Beth. Dynamic rescheduling that simultaneously considers efficiency and stability. *Computers & Industrial Engineering*. 2004, vol. 46, no. 1, pp. 1–15. ISBN 0360-8352. Available from DOI: https://doi.org/10.1016/j.cie.2003.09.007.

48. CARVALHO, Thyago P.; SOARES, Fabrízzio A. A. M. N.; VITA, Roberto; FRANCISCO, Roberto da P.; BASTO, João P.; ALCALÁ, Symone G. S. A systematic literature review of machine learning methods applied to predictive maintenance. *Computers & Industrial Engineering*. 2019, vol. 137, p. 106024. ISBN 0360-8352. Available from DOI: https://doi.org/10.1016/j.cie.2019.106024.

49. TESAURO, Gerald. Temporal difference learning and TD-Gammon. *Commun. ACM*. 1995, vol. 38, no. 3, pp. 58–68. ISSN 0001-0782. Available from DOI: 10.1145/203330.203343.

50. CAMPBELL, Murray; HOANE, A. Joseph; HSU, Feng-hsiung. Deep Blue. *Artificial Intelligence*. 2002, vol. 134, no. 1, pp. 57–83. ISBN 0004-3702. Available from DOI: https://doi.org/10.1016/S0004-3702(01)00129-1.

51. MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; GRAVES, Alex; ANTONOGLOU, Ioannis; WIERSTRA, Daan; RIEDMILLER, Martin. *Playing Atari with Deep Reinforcement Learning*. 2013. Available from arXiv: 1312.5602 [cs.LG].

52. SUTTON, Richard S.; BARTO, Andrew G. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN 0262039249.

53. BOUTILIER, Craig. Planning, learning and coordination in multiagent decision processes. In: *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge*. The Netherlands: Morgan Kaufmann Publishers Inc., 1996, pp. 195–210. TARK '96. ISBN 1558604179.

54. BELLMAN, RICHARD. A Markovian Decision Process. *Journal of Mathematics and Mechanics* [online]. 1957, vol. 6, no. 5, pp. 679–684 [visited on 2024-04-19]. ISSN 00959057, ISSN 19435274. Available from: http://www.jstor.org/stable/24900506.

55. WATKINS, Christopher J. C. H.; DAYAN, Peter. Q-learning. *Machine Learning*. 1992, vol. 8, no. 3, pp. 279–292. ISBN 1573-0565. Available from DOI: 10.1007/BF00992698.

56. HASSELT, Hado. Double Q-learning. In: LAFFERTY, J.; WILLIAMS, C.; SHAWE-TAYLOR, J.; ZEMEL, R.; CULOTTA, A. (eds.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2010, vol. 23. Available also from: https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

57. WILLIAMS, Ronald J.; BAIRD, Leemon C. Analysis of Some Incremental Variants of Policy Iteration: First Steps Toward Understanding Actor-Cr. In: 1993. Available also from: https://api.semanticscholar.org/CorpusID:18786951.

58. HESSEL, Matteo; MODAYIL, Joseph; HASSELT, Hado van; SCHAUL, Tom; OSTROVSKI, Georg; DABNEY, Will; HORGAN, Dan; PIOT, Bilal; AZAR, Mohammad; SILVER, David. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. 2017. Available from arXiv: 1710.02298 [cs.AI].

59. HASSELT, Hado van; GUEZ, Arthur; SILVER, David. *Deep Reinforcement Learning with Double Q-learning*. 2015. Available from arXiv: 1509.06461 [cs.LG].

60. WANG, Ziyu; SCHAUL, Tom; HESSEL, Matteo; HASSELT, Hado van; LANCTOT, Marc; FREITAS, Nando de. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. Available from arXiv: 1511.06581 [cs.LG].

61. SCHAUL, Tom; QUAN, John; ANTONOGLOU, Ioannis; SILVER, David. *Prioritized Experience Replay*. 2016. Available from arXiv: 1511.05952 [cs.LG].

62. FORTUNATO, Meire; AZAR, Mohammad Gheshlaghi; PIOT, Bilal; MENICK, Jacob; OSBAND, Ian; GRAVES, Alex; MNIH, Vlad; MUNOS, Remi; HASSABIS, Demis; PIETQUIN, Olivier; BLUNDELL, Charles; LEGG, Shane. *Noisy Networks for Exploration*. 2019. Available from arXiv: 1706.10295 [cs.LG].

63. PENG, Jing; WILLIAMS, Ronald J. Incremental multi-step Q-learning. *Machine Learning*. 1996, vol. 22, no. 1, pp. 283–290. ISBN 1573-0565. Available from DOI: 10.1007/BF00114731.

64. SCHULMAN, John; MORITZ, Philipp; LEVINE, Sergey; JORDAN, Michael; ABBEEL, Pieter. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. Available from arXiv: 1506.02438 [cs.LG].

65. GREENSMITH, Evan; BARTLETT, Peter; BAXTER, Jonathan. Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning. In: DIETTERICH, T.; BECKER, S.; GHAHRAMANI, Z. (eds.). *Advances in Neural Information Processing Systems*. MIT Press, 2001, vol. 14. Available also from: https://proceedings.neurips.cc/paper_files/paper/2001/file/584b98aac2dddf59ee2cf19ca4ccb75e-Paper.pdf.

66. SUTTON, Richard S; MCALLESTER, David; SINGH, Satinder; MANSOUR, Yishay. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: SOLLA, S.; LEEN, T.; MÜLLER, K. (eds.). *Advances in Neural Information Processing Systems*. MIT Press, 1999, vol. 12. Available also from: https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

67. WEAVER, Lex; TAO, Nigel. *The Optimal Reward Baseline for Gradient-Based Reinforcement Learning*. 2013. Available from arXiv: 1301.2315 [cs.LG].

68. KAKADE, Sham M. A Natural Policy Gradient. In: DIETTERICH, T.; BECKER, S.; GHAHRAMANI, Z. (eds.). *Advances in Neural Information Processing Systems*. MIT Press, 2001, vol. 14. Available also from: https://proceedings.neurips.cc/paper_files/paper/2001/file/4b86abe48d358ecf194c56c69108433e-Paper.pdf.

69. SCHULMAN, John; LEVINE, Sergey; MORITZ, Philipp; JORDAN, Michael I.; ABBEEL, Pieter. *Trust Region Policy Optimization*. 2017. Available from arXiv: 1502.05477 [cs.LG].

70. SCHULMAN, John; WOLSKI, Filip; DHARIWAL, Prafulla; RADFORD, Alec; KLIMOV, Oleg. *Proximal Policy Optimization Algorithms*. 2017. Available from arXiv: 1707.06347 [cs.LG].

71. *The 37 Implementation Details of Proximal Policy Optimization* [online]. [N.d.]. [visited on 2024-04-01]. Available from: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

72. FISH, Amid. *Lessons Learned Reproducing a Deep Reinforcement Learning Paper* [online]. 2018. [visited on 2024-04-01]. Available from: http://amid.fish/reproducing-deep-rl.

73. *Creating Message Passing Networks pytorch$_g$eometricdocumentation*. [N.d.]. Available also from: https://pytorch-geometric.readthedocs.io/en/latest/tutorial/create_gnn.html.

74. LIU, Chuang; ZHAN, Yibing; WU, Jia; LI, Chang; DU, Bo; HU, Wenbin; LIU, Tongliang; TAO, Dacheng. *Graph Pooling for Graph Neural Networks: Progress, Challenges, and Opportunities*. 2023. Available from arXiv: 2204.07321 [cs.LG].

75. DOUGLAS, B. L. *The Weisfeiler-Lehman Method and Graph Isomorphism Testing*. 2011. Available from arXiv: 1101.5211 [math.CO].

76. KIPF, Thomas N.; WELLING, Max. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. Available from arXiv: 1609.02907 [cs.LG].

77. XU, Keyulu; LI, Chengtao; TIAN, Yonglong; SONOBE, Tomohiro; KAWARABAYASHI, Ken-ichi; JEGELKA, Stefanie. *Representation Learning on Graphs with Jumping Knowledge Networks*. 2018. Available from arXiv: 1806.03536 [cs.LG].

78. TKIPF. *Directed graph & edge classification, issue 63, TKIPF/GCN*. [N.d.]. Available also from: https://github.com/tkipf/gcn/issues/63.

79. HAMILTON, William L.; YING, Rex; LESKOVEC, Jure. *Inductive Representation Learning on Large Graphs*. 2018. Available from arXiv: 1706.02216 [cs.SI].

80. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-Term Memory. *Neural Comput.* 1997, vol. 9, no. 8, pp. 1735–1780. ISSN 0899-7667. Available from DOI: 10.1162/neco.1997.9.8.1735.

81. VELIČKOVIĆ, Petar; CUCURULL, Guillem; CASANOVA, Arantxa; ROMERO, Adriana; LIÒ, Pietro; BENGIO, Yoshua. *Graph Attention Networks*. 2018. Available from arXiv: 1710.10903 [stat.ML].

82. BAHDANAU, Dzmitry; CHO, Kyunghyun; BENGIO, Yoshua. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. Available from arXiv: 1409.0473 [cs.CL].

83. MAAS, Andrew L. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In: 2013. Available also from: https://api.semanticscholar.org/CorpusID:16489696.

84. BRODY, Shaked; ALON, Uri; YAHAV, Eran. *How Attentive are Graph Attention Networks?* 2022. Available from arXiv: 2105.14491 [cs.LG].

85. XU, Keyulu; HU, Weihua; LESKOVEC, Jure; JEGELKA, Stefanie. *How Powerful are Graph Neural Networks?* 2019. Available from arXiv: 1810.00826 [cs.LG].

86. ZHANG, Cong; SONG, Wen; CAO, Zhiguang; ZHANG, Jie; TAN, Puay Siew; XU, Chi. *Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning*. 2020. Available from arXiv: 2010.12367 [cs.LG].

87. YANG, Zhong; BI, Li; JIAO, Xiaogang. Combining Reinforcement Learning Algorithms with Graph Neural Networks to Solve Dynamic Job Shop Scheduling Problems. *Processes*. 2023, vol. 11, no. 5. ISBN 2227-9717. Available from DOI: 10.3390/pr11051571.

88. ZHANG, Lu; FENG, Yi; XIAO, Qinge; XU, Yunlang; LI, Di; YANG, Dongsheng; YANG, Zhile. Deep reinforcement learning for dynamic flexible job shop scheduling problem considering variable processing times. *Journal of Manufacturing Systems*. 2023, vol. 71, pp. 257–273. ISBN 0278-6125. Available from DOI: https://doi.org/10.1016/j.jmsy.2023.09.009.

89. LIU, Chien-Liang; HUANG, Tzu-Hsuan. Dynamic Job-Shop Scheduling Problems Using Graph Neural Network and Deep Reinforcement Learning. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. 2023, vol. 53, no. 11, pp. 6836–6848. Available from DOI: 10.1109/TSMC.2023.3287655.

90. LIU, Zhenyu; MAO, Haoyang; SA, Guodong; LIU, Hui; TAN, Jianrong. Dynamic job-shop scheduling using graph reinforcement learning with auxiliary strategy. *Journal of Manufacturing Systems*. 2024, vol. 73, pp. 1–18. ISBN 0278-6125. Available from DOI: https://doi.org/10.1016/j.jmsy.2024.01.002.

91. WANG, Yuhui; HE, Hao; WEN, Chao; TAN, Xiaoyang. *Truly Proximal Policy Optimization*. 2020. Available from arXiv: 1903.07940 [cs.LG].

92. SONG, Wen; CHEN, Xinyang; LI, Qiqiang; CAO, Zhiguang. Flexible Job-Shop Scheduling via Graph Neural Network and Deep Reinforcement Learning. *IEEE Transactions on Industrial Informatics*. 2023, vol. 19, no. 2, pp. 1600–1610. Available from DOI: 10.1109/TII.2022.3189725.

93. MNIH, Volodymyr; BADIA, Adrià Puigdomènech; MIRZA, Mehdi; GRAVES, Alex; LILLICRAP, Timothy P.; HARLEY, Tim; SILVER, David; KAVUKCUOGLU, Koray. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. Available from arXiv: 1602.01783 [cs.LG].

94. HAYEU, Yury. *Dynamic Job Shop Scheduling Simulator*. 2024. Version 1.0.0. Available also from: https://github.com/yura-hb/diploma-thesis/.

95. I, Salvador; HERRERA, Francisco. An Extension on "Statistical Comparisons of Classifiers over Multiple Data Sets" for all Pairwise Comparisons. *Journal of Machine Learning Research - JMLR*. 2008, vol. 9.

96. SHAFFER, Juliet. Multiple Hypothesis Testing. *Annual Review of Psychology*. 1995, vol. 46, pp. 561–84. Available from DOI: 10.1146/annurev.ps.46.020195.003021.

97. KINGMA, Diederik P.; BA, Jimmy. *Adam: A Method for Stochastic Optimization*. 2017. Available from arXiv: 1412.6980 [cs.LG].

98. LOSHCHILOV, Ilya; HUTTER, Frank. *Decoupled Weight Decay Regularization*. 2019. Available from arXiv: 1711.05101 [cs.LG].

99. LIU, Liyuan; JIANG, Haoming; HE, Pengcheng; CHEN, Weizhu; LIU, Xiaodong; GAO, Jianfeng; HAN, Jiawei. *On the Variance of the Adaptive Learning Rate and Beyond*. 2021. Available from arXiv: 1908.03265 [cs.LG].

100. DOHARE, Shibhansh; LAN, Qingfeng; MAHMOOD, A. Rupam. Overcoming Policy Collapse in Deep Reinforcement Learning. In: *Sixteenth European Workshop on Reinforcement Learning*. 2023. Available also from: https://openreview.net/forum?id=m9Jfdz4ym0.

101. GLOROT, Xavier; BENGIO, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In: TEH, Yee Whye; TITTERINGTON, Mike (eds.). *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, vol. 9, pp. 249–256. Proceedings of Machine Learning Research. Available also from: https://proceedings.mlr.press/v9/glorot10a.html.

102. HU, Wei; XIAO, Lechao; PENNINGTON, Jeffrey. *Provable Benefit of Orthogonal Initialization in Optimizing Deep Linear Networks*. 2020. Available from arXiv: 2001.05992 [cs.LG].