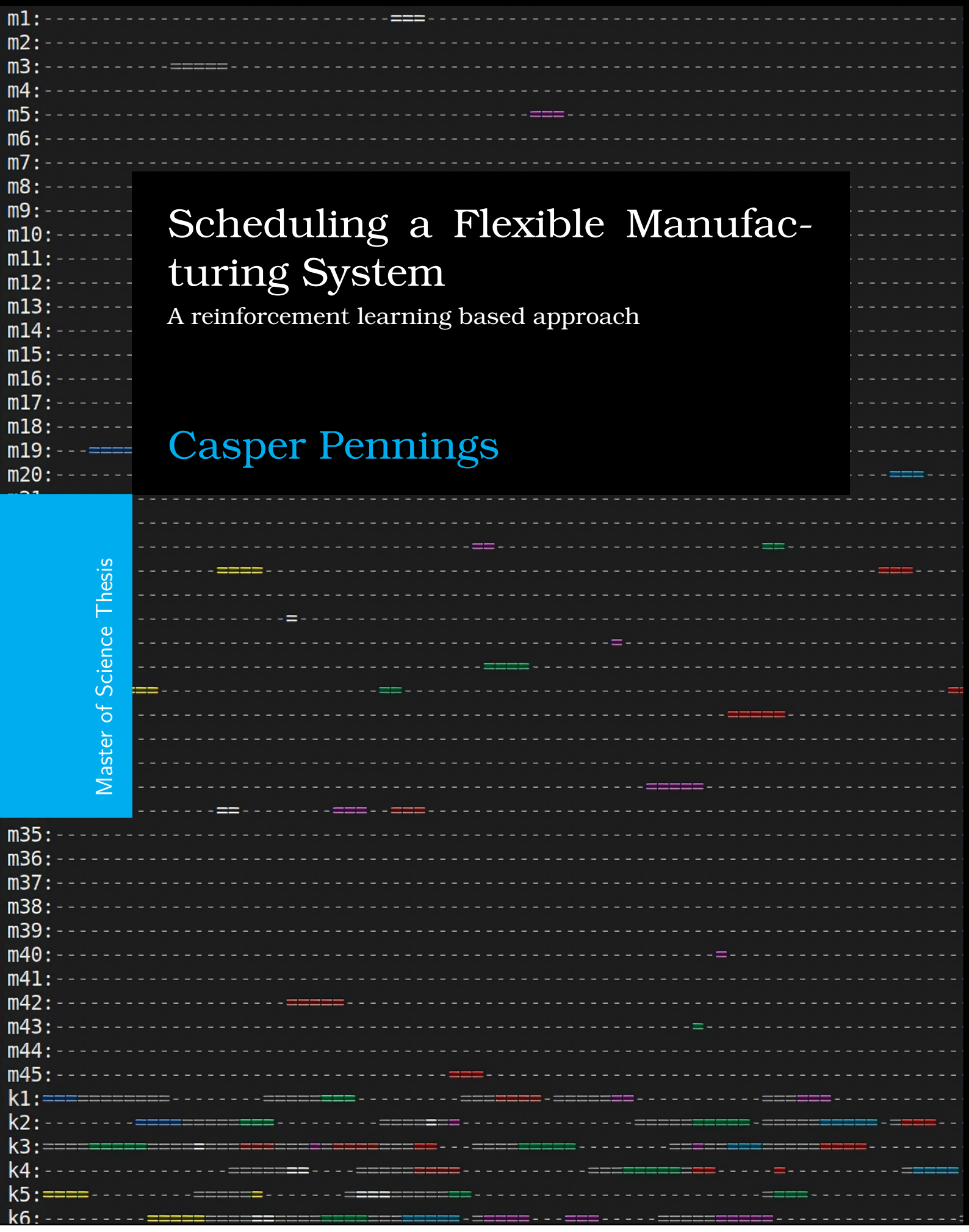

Scheduling a Flexible Manufacturing System

A reinforcement learning based approach

Casper Pennings

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Scheduling a Flexible Manufacturing System

A reinforcement learning based approach

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Casper Pennings

May 28, 2023

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

A flexible manufacturing system (FMS) has advantages over traditional manufacturing systems due to its ability to deal with unpredicted circumstances such as changes in demand or component breakdowns by re-routing. However, this flexibility increases the complexity of controlling such a system. Traditionally, the system model is simplified to reduce the solution space by removing intra-machine transportation complexities. This thesis explores how these complexities can be kept and accounted for during scheduling. A scheme is used where short term schedules are continuously calculated to determine the optimal schedule over the next timeframe. The flexible job shop scheduling problem with transport (FJSPT) is used to represent the complexities of the FMS. To calculate part-schedules repeatedly a fast constructive search method is needed, the AlphaZero framework is identified as a fitting candidate. The FJSPT is translated into the reinforcement learning framework using a reduced action space, a graph neural network based state representation and normalized reward function. A naive normalization approach for the reward function is found to introduce problems in the value function sensitivity, while other adaptive method show fundamental flaws. A novel normalization method is introduced using min-max adaptive normalisation and suboptimal node inclusion to improve value function training data. Implementing and training the algorithm shows the method performs poorly in comparison to metaheuristic based algorithms for the FJSPT problem. The value function is not able to converge to training data, while this is critical for the self-improvement training of the algorithm. Future work should focus on developing a normalized value function that is sensitive to solution quality and is able to converge. Despite the challenges, the work provides insights into the complexities of implementing AlphaZero for combinatorial optimization.

Table of Contents

List of Figures

List of Tables

Acknowledgements

The thesis in front of you in the result of my graduation project and concludes my studies in Systems and Control, as well as my time as a student. The graduation process has been a long journey in which I learned a tremendous amount, not only as a student but also on a personal level. With finishing this thesis, I can close this chapter, reflect, and look forward to new challenges in my life armed with this experience. But not before thanking the people that helped me get it done.

First, I would like to thank my supervisors prof.dr.ir. Tamás Keviczky and dr. Neil Yorke-Smith, for their advice, time, and patience. And apart from the feedback during the project, the environment they created has allowed me to work on myself and improve myself as a professional.

Second, I would like to thank the people around me supporting me during the process. My friends and family, providing unconditional support and belief in me. Especially Bram, who has helped me from the beginning of the project in clearing my head and keeping my focus on the important things.

Lastly, I would like to thank all the people that joined me during the many hours spent on this project. The library study group: Wesley, Vincent, Toon, Joanne and Marijn, and my roommate Maarten who spent 50-hour weeks with me at EWI during parts of my project.

This thesis would not be possible without these people.

Delft, University of Technology — Casper Pennings
May 28, 2023

Chapter 1

Introduction

1-1 Background and motivation

Traditional manufacturing systems are static systems designed for continuous mass production of goods. A linear process is used to transform raw product to the intended products. However, if unpredicted circumstances happen such as a sudden change in demand or production process, such static systems are slow to react. The ability of a manufacturing system to absorb problems with affecting the output is referred to as flexibility. Flexible manufacturing systems (FMS) are design to inhibit this property.

There are varying definitions of FMS given through the years. The US National Bureau of Standard have defined FMS as 'an arrangement of machines that were interconnected by means of a transport system and a central computer controls both machines and transport system', describing a manufacturing system consisting of a set of machines and material handlers. Figure 1-1 illustrates such a system considered in this thesis.

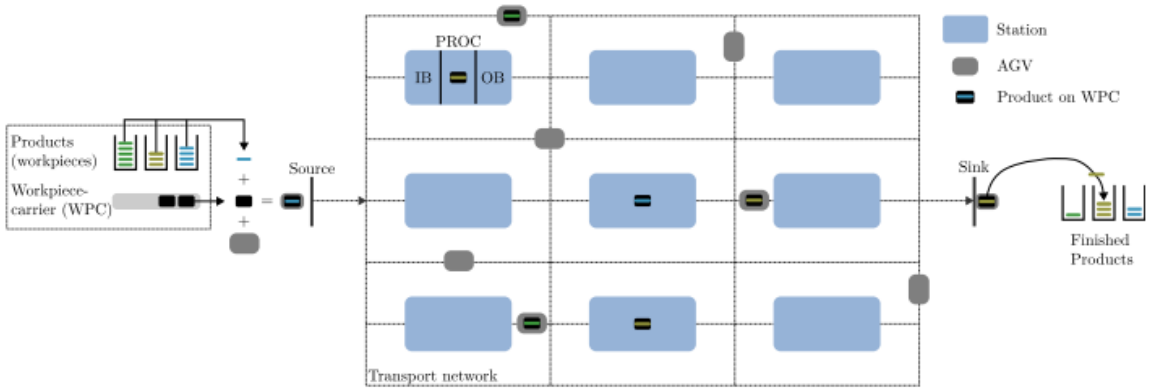

Figure 1-1: Example model of a flexible manufacturing system [1]

By using automated vehicles to transport materials around and multiple machines to carry out operations, a production network is created. As there is flexibility in machine use and

transportation routing, disruptions such a machine failures can be circumvented by adjusting planning. FMS also benefit from ease of scaling as new machines and transportation methods can be added to the production grid.

This higher degree of flexibility, however, leads to the need for optimized decision-making to efficiently process production orders. Scheduling which operations are carried out at what machine at what time, can be solved by optimizing the well researched Flexible Job Scheduling Problem (FJSP). However, the FJSP ignores the transportation time between machines, or sees them as part of the process time per operation. In a realistic FMS, there are a limited number of material handling vehicles, making transportation time dependent on vehicle availability. This introduces additional complexity, which traditionally is ignored [2] for sake of computation time.

Controlling such a system is typically done by *reactive scheduling* [3] determining which action to perform next based on the current state of the system. *Predictive-reactive scheduling* Mayer, Classen, and Endisch [1] instead uses simulation to plan a schedule, when a disruption occurs the schedule is replanned. Calculating a good schedule however is a computationally intensive process as the number of valid schedules quickly scales with the size of the FMS. In order to implement a predictive-reactive scheduling for an FMS including transport time, an algorithm is developed in this thesis to quickly find near optimal schedules. By following these schedules, the system is able to efficiently perform its tasks.

1-2 Scope and Limitations

While there are many FMS which are flexible to differing extends, the literature study will focus on those which consists out of a matrix of machines capable of doing multiple operations connected by single capacity vehicles which supply the machines. Focussing on such a complicated system is done to develop a method capable of working on the most complicated scenarios.

The goal is to construct a schedule for a fixed set of jobs requiring production. This schedule can then be executed while a new schedule is generated. As dynamic scheduling problems generalize to the static scheduling problem solved iteratively [4], the focus is put on offline scheduling. Adapting this solver for online scheduling with predictive-reactive scheduling is left as future work. The main research question is formulated as:

How can near optimal schedules be formed quickly for flexible manufacturing systems?

To structure the research effort into clearer sub-directions, the main question is split into sub-questions.

- What model can be used to represent the complexity and objective of a flexible manufacturing system?
- What kind of solver framework is needed to form the schedules?
- How can this framework be adapted to work on scheduling problems?
- How optimal and quickly is the solver able to form schedules?

1-3 Thesis outline

The thesis is structured as follows:

- Chapter 2 introduces the scheduling problem and solving methods
- Chapter 3 discusses how reinforcement learning can be used to optimize schedules
- Chapter 4 details how the FMS problem was implemented in the reinforcement learning framework
- Chapter 5 discusses the implementation details and the results
- Chapter 6 summarizes the results of the work

Chapter 2

Schedule Optimization

In this chapter, the scheduling problem is introduced. After introducing the problem, the complexities of solving the problem and previous work are discussed.

2-1 Flexible job scheduling problem with transport

As scheduling problems are difficult to optimize, frequently abstractions are used to simplify the problem and limit the solution space. For FMS scheduling problems, in the past this was often done by omitting the transport complexities or optimizing it afterwards. These abstractions however lead to non-optimal solution in real life. By using advanced optimization techniques, this thesis attempts to simultaneously scheduling transportation and machining.

Problem definition

The scheduling problem considered in this thesis is the flexible job shop scheduling problem with transport (FJSPT). The FJSPT is a scheduling problem intended to replicate the complexities of an FMS [5] by not only scheduling the processing operation on the machines but optimizing the transportation as well. The problem can formally be described using the $\alpha|\beta|\gamma$ notation, proposed by Graham, Lawler, Lenstra, *et al.* [6], as the $FJ_m|\text{tr}(v)|c_{\max}$ problem.

Machine environment - α

The machine environment is a flexible job shop environment FJ_m. In a FJ_m environment a set of jobs $i \in J$ each consisting of an ordered set of operations N_i with $n_i > 0$ operations $N_i = \{o_{ij} : j = 1, 2, \ldots, n_i\}$ and $O = \bigcup_{i \in J} N_i$. Each operation o_{ij} is to be scheduled in on one of machines $m \in M$, taking process time p_{ij}^m. A machine can process a single operation at the time.

Constraints - β

The problem is subject to transport constraints $tr(v)$. Transport constraints add the complexity of transporting the materials needed per operation o_{ij} to machine m from previous location m' with transporting operation T_{ij} taking time $t_{ij}^{mm'}$ by a transport vehicle $k \in K$. Where a transport vehicle can only transport a single material unit, which picked up and dropped of before and the after the first and last operation at a load/unload point (LU).

While in a $tr(\infty)$ there are infinite vehicles available and in a $tr(1)$ all transport in done by a single vehicle, in a $tr(v)$ problem any discrete number of vehicles are considered. As there are a limited number of vehicles, each vehicle's current position needs to be considered in determining their transport time.

Objective - γ

The objective is the $c_{\max}$ objective. The goal of the $c_{\max}$ objective is to minimize the time needed to complete all jobs. Formally, each job has completion time c_i where the maximum completion time of all jobs is minimized. The job time consists of the transport time, process time and idle time u_{ij}^{km} for each operation plus a dummy operation for transport back to the LU point.

$$c_i = \sum_{i=0}^{n(i)+1} t_{ij}^{mm'} + p_{ij}^{m} + u_{ij}^{km} \tag{2-1}$$

$$c_{\max} = \max_{i \in J} c_i \tag{2-2}$$

$$\min c_{\max} \tag{2-3}$$

A solution to the FJSPT consists of four parts. The vehicle assignment $O_k(o_{ij}) : O \to K$ indicating which vehicle is assigned to each operation, and equivalently the machine assignment $O_m(o_{ij}) : O \to M$. Additionally, the transport sequence $S_t = \{t_1, \ldots, t_{|O|}\}$ and the process sequence $S_p = \{p_1, \ldots, p_{|O|}\}$ indicate the order of transport and process operations. A mixed integer linear programming (MILP) formulation of the FJSPT problem is formulated in [7].

To find good schedules for the FJSPT, a combinatorial search is to be used. Chapt. 2-2 introduces combinatorial optimization and discusses its complexities.

2-2 Combinatorial optimization

2-2-1 Definition

Combinatorial optimization is a subfield of mathematical optimization. In mathematical optimization, a problem can be either continuous or discrete. Given a set X, a function $f : X \to \mathbb{R}$ and a subset I of X we consider the problem:

$$\min f(x) \quad \text{s.t.} \quad x \in I$$

Continuous problems require all variables in the objective function f to be continuous variables, that is, real values with no gaps. Discrete optimization allows some or all variables

to be discrete values, such as integers. While combinatorial problems can also have both continuous and discrete variables, we only consider those with a finite set of solutions and thus only discrete variables.

2-2-2 Complexity

While solving a combinatorial problem may seem more trivial than a continuous problem, as combinatorial have a limited search space, the opposite is true. Continuous problems can use derivatives and gradients to describe the slope of a function, but combinatorial problems do not. Thus, alternative, less efficient methods are needed to explore the solution space.

The solution space S for combinatorial problems is often very large, due to S typically scaling at least exponentially with the size of the problem [8]. For example, for a *travelling salesmen problem* with 15 cities, there are 43,589,145,600 candidate solutions, eliminating the possibility of brute force search methods.

A large solution space however does not make a combinatorial problem hard to solve. A well known example is the *shortest path problem*, which has an exponential large solution space, but can be solved in quadratic time [9]. To differentiate between these problems, complexity classes can be used.

P Problems belonging to the P class are solvable by a deterministic machine in polynomial time w.r.t. the size. The computation time for P problems, such as the shortest path problem, scale less rapid than problems which for example need exponential time.

NP NP problems on the other hand need a nondeterministic machine to be solved in polynomial time. A nondeterministic machine in this context is a hypothetical machine which able to make correct guesses for certain decisions [8]. In the real world, exponential the best algorithms known so far need exponential time to find a solution. It is unknown if P and NP are the same class of problem for which NP problems lack an efficient solution, in case it is not, P is a subclass of NP.

NP-Hard NP-hard problems are problems which are at least as hard as NP problems, due to all NP problems being able to be reduced to NP-hard problems in polynomial time. Thus, if a NP-hard problem would be able to be solved in polynomial time on a deterministic machine, all NP problems would be. NP-hard problems may have exponential complexity, in which case they are part of NP and referred to as NP-complete, or even larger complexity.

2-2-3 Methods

Solving combinatorial problems require searching their solution space. Methods for searching can be explained by combining two search paradigms, constructive search and perturbative search [8]. Both paradigms work using the fact that combinatorial solutions are composed of solution components which together describe the solution of the problem.

Constructive search Iteratively extends partial solutions trying to form a good solution, methods vary in the way the extension is chosen

Perturbative search Iteratively changes a solution by modifying one or more solution components to form a new solution, methods vary in the way perturbations are chosen

Using these paradigms, two classes of search algorithms can be formed.

Systematic search

By combining constructive search with a backtracking method, systematic search methods are formed. A solution is build by iteratively extending it with partial solutions until a complete solution is formed, by then backtracking to the most recent choice point with unexplored choices alternative solutions are explored. The process can be visualized as a tree, of which its branches get explored. Such a method would be able to find all optimal solutions and would therefore be *complete*, however it would need at least exponential run time to explore all solutions. Two methods can be used to reduce the computation time.

Pruning excludes branches from the solution space if they will not result in a better solution. For problems where a lower bound can be quickly computed for partial solutions, a branch can be discarded if the lower bound of an alternative choice is worse than the best found solution.

Heuristics guide the search in directions which will lead to good solutions. Heuristics quickly evaluate part solutions to find which direction will potentially lead to the best solution. If the heuristic indicates a better solution can be found than the current best known solution, that direction will be explored. As solutions that the heuristic indicates will lead to worse solutions are not explored, the solution space is cut down. A heuristic that is always an underestimation of the real value of the solution is called *admissible*, and will result in finding an optimal solution.

Using these methods, a balance can be made between optimality and speed. When a non-admissible heuristic is used or branches are aggressively pruned, good solutions will be found in less time due to the search being narrower. However, no guarantees can be given on optimality, making the algorithm an approximate algorithm. By balancing speed with optimality, systematic search can be tuned.

Local search

By combining constructive search and perturbative search, local search methods are formed. Local search first builds a solution by extending partial solutions with constructive search until a candidate solution is formed, before exploring its neighbourhood via perturbative search. By continuously adjusting solution components, the candidate solution is improved. A problem however occurs when the candidate solution reaches a solution for which perturbations lead to worse solutions, but is not optimal, a local optima. There are multiple ways to deal with these local optima.

The most trivial approach is the *restart strategy* [8], when a (local) optimum is encountered the solution is stored and the algorithm is reinitialised from a different initial candidate solution. An alternative strategy is to perform a non-improving step when a local optima is encountered to escape the optima.

2-2-4 Optimizing the FJSPT

With the FJSPT defined and the complexities of combinatorial optimization known, an approach is chosen for optimizing the FJPST.

Complexity

Optimizing the FJSPT is non-trivial, as it introduces multiple challenges.

The FJSPT is a generalization of the job scheduling problem (JSP) which has been proven to be NP-hard [10], making the FJSPT NP-hard. Being an NP-hard combinatorial problem, a combinatorial search is needed over the solution space I. However, due to the added flexibility in machines, vehicles and transport requests, the FJSPT suffers from combinatorial explosion. The solution space grows over the number of jobs J, average number of machines available $\bar{A}$ and vehicles K, an estimation of the solution space I can be made ignoring the number of jobs $|J|$ decreasing when jobs are fully scheduled.

$$|I| \approx \Big(\underbrace{|J| \cdot |K|}_{\text{Transport options}} * \underbrace{|J| \cdot |\bar{A}|}_{\text{Process options}} \Big)^{|O|} \tag{2-4}$$

As the problem can be seen as a sequential decision process where for each operation a job vehicle and job machine option can be chosen.

Due to the size exponentially growing with the number of operations, an efficient combinatorial search algorithm is needed to find solutions in problems with a non-small number of operations.

Further complexity is introduced by the choice of objective. While some common objectives can be decomposed into smaller objectives, such as tardiness or lateness [11] can be decomposed per job, the makespan can not be due to the maximization in (2-2). This leads to a sparse reward function in a RL setting. Furthermore, it is not possible to determine the effect of scheduling a single operation on the final makespan, which compromises the ability to form an efficient search heuristic.

Rolling horizon scheduling

As the end goal is to find schedules used in predictive-reactive scheduling, a rolling horizon approach can be used to decrease the search space. As implemented in [12], by only scheduling a number of operations ahead, a partial schedule can be generated instead. By continuously expanding this partial schedule online, a full schedule is generated. In order to use such a scheme, a systematic search method should be used to continually extend partial solutions.

Previous work on FJSPT

Previous approaches for the FJSPT problem have been summarized chronologically in Table 2-1. For each part of the solution, the optimization method or heuristic in italics is given.

Different types of metaheuristic algorithms were used to find solutions, differential evolution (DE), genetic algorithms (GE), tabu search (TS), particle swarm optimization (PSO) and

late acceptance hill climbing (LAHC). Detail about these algorithms can be found in the corresponding papers.

In all but one attempt, heuristics were used to decrease the size of the solution space. A downside of using heuristics is that optimal solution might be left out of the search space. However, the LAHC approach [7] was able to find optimal solutions even when using the same order for the transport and process operations, thus the heuristic might not exclude optimal solutions. However, no proof was provided in the paper.

All but one approach made use of a local search method. Only [13] made use of a systematic search method, however this method only used heuristics to form the schedule and did not find competitive solutions.

Table 2-1: Solution components FJSPT methods, optimized components are optimized simultaneously while others are determined from the solution via heuristics

Author	S_p	S_t	O_m	O_k
Kumar et al. [14]	DE	S_p	*Greedy*	*Greedy*
Zhang et al. [15]	GA+TS	*Operation finish time*	GA+TS	GA+TS
Zhang et al. [16]	*Shifting bottleneck*	*Shifting bottleneck*	GA+TS	GA+TS
Zhang et al. [13]	*Shifting bottleneck*	*Greedy*	*Shifting bottleneck*	*Greedy*
Deroussi [17]	*FIFO*	PSO+SLS	*Greedy*	PSO+SLS
Nouri et al. [18]	GA+TS	GA+TS	GA+TS	GA+TS
Homayouni, Fontes [7]	LAHC	S_p	LAHC	LAHC

No competitive constructive search algorithm was designed for the FJSPT problem so far. Finding good solutions using a constructive search algorithm for the FJSPT requires exploring promising regions. A good heuristic can point the search in these regions. Such a heuristic can be formed by hand, however a recent approach is to use reinforcement learning to form the heuristic. The next chapter will introduce reinforcement learning based optimization.

Chapter 3

Reinforcement Learning Based Optimization

As described in chapter 2, online scheduling of CO problems requires finding good solutions in the solution space quickly. Heuristics and pruning can be used to guide the solver. To do so, expert knowledge can be used to set up rule based heuristics and pruning, however forming performant rules is non-trivial and can become infeasible with increasing model complexity. Using a Reinforcement Learning (RL) framework, this heuristic can be approximated instead by fitting it to simulation results.

RL is a machine learning paradigm based on solving sequential decision problems. RL differs from other machine learning paradigms, such as supervised and unsupervised learning, as it learns based on exploratory interactions with the environment instead of a supplied set of data, mimicking a trial-and-error approach. By iteratively reinforcing desired behaviour via rewards, a modelled agent is optimized to choose actions which maximize the future rewards. By linking rewards with the optimality of each decision, RL can learn to solve decision problems.

3-1 Reinforcement Learning Framework

In order to solve the CO problem using RL, the problem is formulated in a RL framework. When formulated correctly, solving the RL problem will solve the CO problem.

3-1-1 Markov Decision Process

A Markov decision process (MDP) [19] is a mathematical framework for decision problems, which can be used to model most RL problems. An MDP models the environment as a set of states $s \in \mathcal{S}$ which are connected by actions $a \in \mathcal{A}$. An action a will lead to a transition from

the current state s to a new state s' with probability distribution defined by the transition probability function $\mathcal{P} : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \to [0, 1]$.

$$\mathcal{P}(s, s', a) = \Pr(s_{t+1} = s' | s_t = s, a_t = a) \tag{3-1}$$

For each state action combination, a reward r is defined by the reward function $R : (\mathcal{S} \times \mathcal{A}) \to \mathbb{R}$. The sum of all rewards gathered over time is G_t, a discount factor $\gamma \in [0, 1]$ is introduced to keep G_t finite.

$$G_t(s, a) = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | s, a \tag{3-2}$$

The MDP can be described by the 5-tuple $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$.

In order to model a problem as an MDP, two conditions need to be satisfied. Firstly, the *Markov property*, which states that the transition probability distribution should only depend on the current state and action. Enforcing that all relevant information is included in the state and action design.

Secondly, the environment should be *fully observable*, meaning that all information about the state is available at all times (perfect-information). Otherwise, a partially observable Markov decision process (POMDP) [20] framework is needed.

3-1-2 Policies

The decision behaviour of the agent is defined through the policy function. A policy defines a map between states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$ to define behaviour $\pi(s, a) : \mathcal{S} \times \mathcal{A} \to [0, 1]$ or vector $\pi(s) : \mathcal{S} \to \mathcal{A}$. Due to the Markov property, which ensures a state holds all information to determine futures states, the state is sufficient information to determine the optimal action [21].

With a given policy, the MDP can be simplified into a Markov chain as the action can be completely determined from the state, eliminating it from the transition probability function.

$$\mathcal{P}^{\pi}(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = \pi(s)) \tag{3-3}$$

Allowing calculation of the expected rewards starting from state s following policy π, called the value function of π or V-value. Indicating the expected long-term results of being in a state.

$$V^{\pi}(s) = \mathbb{E}\Big[G_t^{\pi}(s, a = \pi(s))\Big] \tag{3-4}$$

And the expected rewards after taking action a in state s and following policy π afterwards, called the action-value function or Q-function. Indicating the expected long-term results of taking an action in a state.

$$Q^{\pi}(s, a) = \mathbb{E}\Big[G_t^{\pi}(s, a)\Big] \tag{3-5}$$

The optimal policy for each state can be defined as the action that maximizes the action-value function, assuming subsequent steps follow an optimal policy.

$$V^*(s) = \max_{\pi} V(s) \quad \forall s \in \mathcal{S} \tag{3-6}$$

$$Q^*(s) = \max_{\pi} Q(s,a) \quad \forall s \in \mathcal{S} \quad \forall a \in \mathcal{A} \tag{3-7}$$

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^{\pi^*}(s,a) \tag{3-8}$$

3-2 Generalized policy iteration

Optimizing the policy requires finding optimal policy π^* maximizing value functions $V^\pi(s)$ or $Q^\pi(s,a)$. The value functions however are not known and need to be estimated from the current policy based either on a defined or learned model of the environment (model-based methods), or on interactions with the environment (model-free methods).

Based on the estimated value functions, the policy can subsequently be improved, leading to a new improved policy. This process is called the generalized policy iteration (GPI) and can be used to describe almost all reinforcement learning methods [20]. The two interacting steps of GPI are:

1. *Policy evaluation*, which estimates $V^\pi(s)$ or $Q^\pi(s,a)$ from current policy π
2. *Policy improvement*, which optimizes current policy π to improved policy π' from estimated $V^\pi(s)$ or $Q^\pi(s,a)$

3-2-1 Model-free policy optimization

Model-free methods optimize the policy purely on interactions with the environment. By optimizing the policy using these interactions, a policy is learned to maximize future rewards. A brief overview is given of the workings of model free methods.

Value-based methods such as Q-Learning [22] estimate $Q(s,a)$ by updating it repeatedly based on received rewards as the policy evaluation step.

$$Q_{k+1}(s_t,a_t) \leftarrow Q_k(s_t,a_t) + \alpha\Big(R(s_t,a_t) + \gamma \max_{a_{t+1} \in \mathcal{A}} Q(s_{t+1},a_{t+1}) - Q_k(s_t,a_t)\Big) \tag{3-9}$$

The policy improvement the uses the updated Q function to greedily select the next action.

$$\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s,a) \tag{3-10}$$

Policy-based methods instead parameterize the policy $\pi(s,\boldsymbol{\theta})$ and use a gradient descent step, with step size α, to optimize the policy in a direction that maximizes the value function.

$$\begin{aligned} J(\boldsymbol{\theta}) &= V^{\pi_{\boldsymbol{\theta}}}(s_0) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t) \end{aligned} \tag{3-11}$$

Determining the gradient $\nabla J(\boldsymbol{\theta}_t)$ can be seen as the policy evaluation step, while the policy improvement step is made using the gradient descent step. Methods for estimating the gradient are outside the scope of this thesis, for information the reader is referred to [20], [23]–[26].

3-2-2 Model-based policy optimization

Model-based methods, in contrast to model-free methods, make use of a model of the environment to make predictions of the effect of suggested actions. This model, formally probability function $\mathcal{P}(s, s', a)$, can be either constructed manually or even learned from the environment [27]–[29].

Planning-based methods can use this model to forward simulate the value function for a certain policy $V^{\pi}(s)$ by following this policy until a terminal state is reached. By not only evaluating current policy π but also nearby policies, making slightly different choices, multiple policies are evaluated during the policy evaluation step. Policy improvement then updates the policy based on the simulated value functions of the neighbouring solution. By using the updated policy in a new environment, new neighbouring policies can be evaluated as the next policy evaluation step.

Model-based methods can be used in a CO setting, as a perfect model of the environment is known anyway as it defines the optimization problem. Model-based methods offer several advantages over model-free methods [30], such as less or no interactions needed with the real world. Another advantage is that the policy evaluation step can be used online, not only exploring the best solution found using the current policy, but also of nearby policies.

3-3 Planning Based Policy Optimization for CO

Building on the RL framework overview, the Expert Iteration (ExIt)/AlphaZero framework is introduced. This framework was introduced independently by both Deepmind as AlphaGo/AlphaZero [31] and as ExIt by Anthony, Tian, and Barber [32]. First, the algorithm is explained, after which it is described how it can be used to solve CO problems.

3-3-1 AlphaZero / Expert Iteration

The AlphaZero (AZ) algorithm is a result of the long-standing challenge of creating an AI for the game Go. Go is a two player board game where the players try to encircle the opponent's pieces by surrounding it by their own pieces. Due to the player being allowed to place a piece on any empty spot on the 19x19 board, there are many possible actions per turn. This results in an enormous solution space for AI methods, rendering brute-force methods ineffective.

The initial version of AZ, AlphaGo [33] became the first computer program to defeat a professional Go player. It was first trained on playing data to mimic a professional Go player, after which it was improved using planning-based RL. AZ was a follow-up of this algorithm, relying purely on planning-based RL outperforming AlphaGo. Independently, ExIt was developed by [32] differing only slightly from AZ. Due to the popularity of AlphaZero, both methods which be referred to as AZ.

Monte Carlo Tree Search Exploration

AZ is a model-based policy optimization method that uses Monte Carlo Tree Search (MCTS) as planning framework. The MDP problem is transformed to a tree graph, where each state

corresponds to a node and each edge to an action. Traversing the tree corresponds to exploring a certain sequence of actions leading to a state in the future. The tree graph is iteratively explored by traversing from the root node, sampling different paths of the tree graph. Each iteration is described in Figure 3-1.

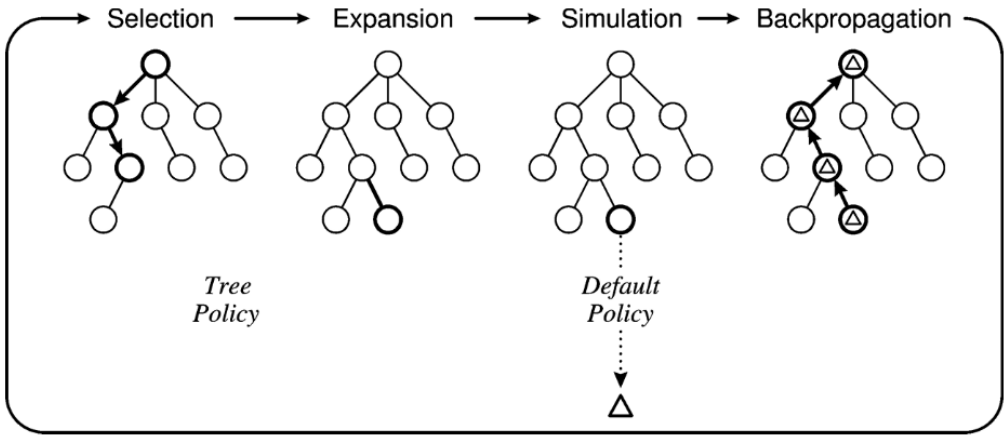

Figure 3-1: Single iteration of the general MTCS method [34]

Each iteration consists of four steps:

1. **Selection**: From the root node, a child selection policy is recursively applied until an unexpanded node is reached.
2. **Expansion**: A new child node is added to the tree according to the available actions.
3. **Simulation**: A child node is evaluated according to a default policy by simulating until a terminal node is reached.
4. **Backpropagation**: The result from the simulation is propagated back through the network to update the involved nodes.

Guided by two policies:

1. **Tree policy**: Selects leaf child node to either explore or create (selection and expansion). The tree policy is key as it determines which states are explored. The most common tree policy is the upper confidence bounds for trees (**UCT**) algorithm [35]. Where for each state s action a is chosen via:

$$UCT(s) = \arg\max_{a \in \mathcal{A}} \underbrace{\frac{\sum_{i=1}^{n(s,a)} R_i(s,a)}{n(s,a)}}_{\text{exploitation}} + \underbrace{c\sqrt{\frac{\ln(\sum_a n(s,a))}{n(s,a)}}}_{\text{exploration}} \tag{3-12}$$

   Where $n(s,a)$ is the number of visits to action a from state s, $R_i(s,a)$ the back propagated terminal rewards and c a tuning parameter. The UCT policy works by calculating for each action the average reward received so far when taking this action and a confidence bound determined by the number of times this action was chosen. The sum of

these is the upper bound of the value estimate of the action. In practise, the first term can be seen as the exploitation value, how often regions are explored that have previously seen good results, and the exploration value, how often new regions are explored. Using constant c these can be balanced, known as the exploration-exploitation trade off, which is key in RL algorithms [20].

2. **Default policy**: Select decisions to play out from non-terminal node to terminal node to produce a value estimate, in the simplest case a uniformly random distribution (simulation).

Policy Optimization

To use MCTS for policy optimization, the tree policy is slightly adjusted to incorporate the policy $\pi(s)$.

$$UCT_{\alpha 0}(s) = \arg\max_{a \in \mathcal{A}} \underbrace{\frac{\sum_{i=1}^{n(s,a)} R_i(s,a)}{n(s,a)}}_{\text{exploitation}} + \underbrace{c \cdot \pi(a|s) \cdot \frac{\sqrt{\sum_a n(s,a)}}{1+n(s,a)}}_{\text{exploration}} \tag{3-13}$$

Two modifications are done, the policy is included in the exploration term to guide the search and the exploration fraction is modified to not always choose unexplored actions first. By following the current policy but also explore based on visit count and samples, policies around the current policy are explored. This not only evaluates the current policy, but also nearby policies in policy evaluation step of the GPI. In order to improve the policy, the exploitation property of UCT is used. As the search is guided towards high rewarding regions, these regions will have a higher visit count $n(s,a)$, which can used to construct an improved policy:

$$\pi_{\text{mcts}}(s,a) = \frac{n(s,a)}{\sum_a n(s,a)}$$

As tabular representations of the policy quickly become infeasible for large state spaces, the policy is represented by an artificial neural network $\pi_{\boldsymbol{\theta}}(s)$ (ANN). ANNs are parametric function approximators mimicking the human nerve system. ANNs can "learn" by adjusting its parameters to minimize a penalty function, the loss function.

The policy loss function is formed by the cross-entropy between the current policy and the improved MCTS-policy for the collected samples.

$$L_\pi(\boldsymbol{\theta}) = -\sum_t \pi_{mcts}(s_t) \log\left(\pi_{\boldsymbol{\theta}}(s_t)\right)$$

Additionally, to improve performance, the default policy is replaced by an additional ANN. This ANN is trained to minimize the squared error between the ANN $V_{\boldsymbol{\theta}}(s)$ and the real value of the state evaluated by MCTS.

$$L_V(\boldsymbol{\theta}) = \sum_t \Big(V_{\boldsymbol{\theta}}(s_t) - V_{mcts}(s_t)\Big)^2$$

The total loss function is formed by summing both losses with some additional parameter weight regulation.

$$L(\boldsymbol{\theta}) = L_\pi(\boldsymbol{\theta}) + L_V(\boldsymbol{\theta}) + c||\boldsymbol{\theta}||^2$$

3-3-2 Learning to solve scheduling problems

Past reinforcement learning methods for solving scheduling problems can be categorized by their planning approach. Dispatching based approaches are fully reactive methods that decide for each scheduling decision what the best next action is, while planning based approaches plan out the full schedule based on the current job list.

Dispatching methods

The advantage of dispatching methods are their applicably to online scheduling, as they only evaluate the best next decision they are generally fast and can calculate the decision live. As that decision is based on the current situation, they can react to disturbances in the system. A disadvantage with dispatching methods is that they have difficulty with sparce rewards, as they have to back-propagate rewards, increasing difficulty in efficiently exploring promising area's [36]. As explained in the reward chapter, correlative rewards can be used to mitigate this effect.

Before the dawn of DNN, Q-learning was used by multiple authors [37]–[39] to train a dispatching agent. To deal with limited computational and generalizing performance, the methods instead of deciding the optimal action, the methods relied on rule based action selection methods. These rules, referred to as dispatching or priority rules, are based on fast to compute principles, such as first in first out (FIFO) or shortest processing time (SPT). The Q-learning algorithm decides which of these dispatching rules should be used. By using DQL instead [40], [41] were able to improve performance and generalizability.

For action selection RL methods, PPO has been dominantly the method of choice for recent methods [42]–[46], praised for its efficiency due to directly optimizing the policy and stability due to the stabilizing properties of the objective. It is difficult to compare different RL approaches due to differences in state representation, settings and training time, however the PPO implementation from [44] outperformed [47] by 5% and 10% on two training sets. In a comparison with meta-heuristic methods, the PPO implementation from [45] was able to beat a genetic algorithm in most test instances with a millisecond of execution time, while the genetic algorithm needed 28 seconds to compute its solution, demonstrating the superiority of RL based methods in dispatching.

Planning methods

Planning based methods instead plan out the total schedule. An advantage of planning based methods is that the resulting makespan of scheduling an operation is known, so scheduling an operation will not lead to unforeseen inefficient results. As planning requires a model of the system behaviour, RL approaches are model-based. To the best of our knowledge, two attempts have been made to use model-based RL for job scheduling.

1. Wang et al. [43] implemented AlphaZero based planning for a parallel machine shop. Interestingly, they showed that a PPO implementation outperformed their AlphaZero implementation by achieving better results with lower training times. The work however uses a dense reward function as the objective is to optimize the tardiness. They found

that adding tree search improved the performance for both the AlphaZero and PPO found policy and found their methods to perform better on large instances, getting outperformed by a genetic algorithm on 15×5 and 25×10 problem sets.

2. An implementation [48] for a substype of the JSP for sheet metal, the SM-JSP, was developed based on a less generalized variant of AlphaZero, AlphaGo Zero, which was developed for playing Go. The implementation is pretrained using the earliest due date (EDD) dispatching rule to give the algorithm a starting point. In the development of the AlphaZero method, it was determined that better performance can be reached by not pretraining the network [31]. Performance is not evaluated on different methods other than EDD and pure MCTS, to which it is superior.

There are clear limitations to the current work as [43] work focusses on tardiness instead of makespan and [48] uses a suboptimal implementation and does not evaluate the results with other methods.

3-3-3 AlphaZero based combinatorial optimization

A naive approach would be to directly apply the AZ framework to a combinatorial problem. However, differences between two players zero-sum games and combinatorial problems cause a number of complexities.

State representation

The state s represents the current state of the agent. The state should comply with the Markov property, which assures that it contains all information necessary to determine the next state when combined with a chosen action. In a two player game, the state contains the current state of the game, often times the board state. In a combinatorial problem, however, the amount of information varies on the size of the problem instance. Thus, needing a state representation capable of representing states with different amounts of information.

Rewards

The tree policy of the MCTS can be seen as consisting out of an exploitation and exploration part, as seen in (3-13). Balancing these parts is known as the exploration-exploitation trade-off, a good balance is necessary for RL problems to improve [20]. As a constant is used to balance the parts, the ratio between exploitation and exploration will differ based on the size of the rewards. In a two-player game, the reward is limited between $[-1, 1]$, where 1 is a win and -1 a loss. However, in a CO problem rewards can have varying sizes, without normalization this would skew the exploration-exploitation ratio.

This normalization method however has to be carefully chosen, as it should have the following characteristics.

1. Be consistent between problem instances in grading the quality of a solution

2. Have a high sensitivity of the quality of a solution

When the reward is not normalized consistently between problem instances $V_{\boldsymbol{\theta}}$ can not learn to estimate $V(s)$ right. While a low sensitivity would require a very accurate estimate in order to differentiate between good and bad solutions.

The reinforcement learning framework was introduced, and the AZ algorithm was chosen to be used. There are several complexities with implementing this algorithm for combinatorial problems, the next chapter will discuss how to handle these issues and apply the algorithm to the FJSPT problem.

Chapter 4

Flexible Job Scheduling Problem with Transport

To solve the FJSPT problem using a reinforcement learning based optimization method, it is formulated as an MDP problem. As this formulation has a big influence on the performance of the reinforcement learning problem [20], careful consideration was taken for each part of the MDP model. In the following subchapters, each part is explained and motivated.

4-1 Action space

The action space $\mathcal{A}_s$ defines the actions available from each state of the MDP. By modelling the FJSPT as a sequential decision problem, the action sequence of the RL agent forms a solution to the FJSPT.

4-1-1 Constructing a schedule

We can demonstrate going from a sequence of actions $S_a = \{a_1, \ldots, a_n\}$ to a valid schedule using a simple job shop scheduling problem $J_m||c_{\max}$. To implement a constructive search method, a schedule is built by iteratively scheduling the next operation from a not fully scheduled job. We can define this set as $J_{\text{active}} \subseteq J$, so $\mathcal{A}_s = J_{\text{active}}$ and $|\mathcal{A}_s| \leq |J| \quad \forall s$. When a job is chosen, it's next operation is scheduled as soon as possible while satisfying precedence and occupancy constraints, resulting in completion time $c_{o_{ij}}$. The completion time for a newly scheduled operation in this example can be calculated as the max of the previous operation completion time $c_{o_{i(j-1)}}$ and the completion time of the last scheduled operation on the machine c_m plus the process time. In Figure 4-1 such a schedule is presented.

$$c_{o_{ij}} = \max(c_{o_{ij-1}}, c_m) + p_{ij} \tag{4-1}$$

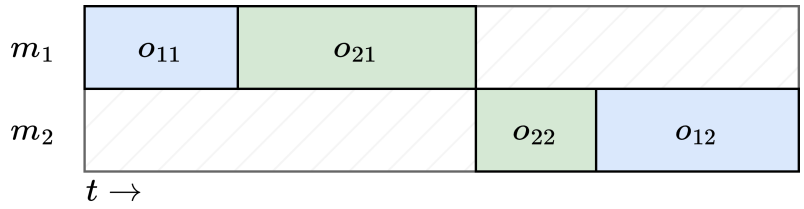

Figure 4-1: Schedule from action sequence $S_a = \{j_1, j_2, j_2, j_1\}$

4-1-2 The FJSPT action space

However, the FJSPT bring several complexities over a simple job shop scheduling problem. Firstly, each operation o_{ij} consists of both a transport t_{ij} and a process step p_{ij}, thus alternatingly being in part of the set of to transport jobs J_T or to process jobs J_P, where $J_{\text{active}} = J_T \cup J_P$. Secondly, each process operation in J_{process} can be processed on any machine capable op the operation $m \in A_{ij}$ and each transport operation can be done using any vehicle $k \in K$. Thus, the actions space becomes

$$\mathcal{A} = \Big(J_T \times K\Big) \cup \Big(J_P \times A\Big) \tag{4-2}$$

Scheduling an operation consist of two parts, the completion time of the transport operation $c_{t_{ij}}$ and the processing step $c_{p_{ij}}$ where $c_{p_{ij}} \geq c_{t_{ij}} \quad \forall i, j$. The completion time of the transportation step depends on the completion time of the previous operation of the job $c_{p_{ij-1}}$, the completion time of the previous operation of the vehicle c_k, the setup time $t^{m_k m'}$ to drive from the machine from t_k: m_k to p_{ij-1}: m' and the drive time $t^{m' m}$ to the machine for processing p_{ij}: m. The processing step processing time p_{ij}^m is expanded to depend on the machine m it is carried out on.

$$c_{t_{ij}} = \max(c_{p_{ij-1}}, c_k + t^{m_k m'}) + t^{m' m} \tag{4-3}$$
$$c_{p_{ij}} = \max(c_{t_{ij}}, c_m) + p_{ij}^m \tag{4-4}$$

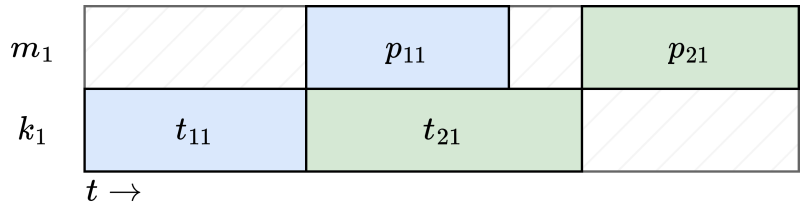

Figure 4-2: Schedule from action sequence $S_a = \{(j_1, k_1), (j_1, m_1), (j_2, k_1), (j_2, m_1)\}$

4-1-3 Reducing the action space

As the combinatorial search happens in the solution space, ideally we would only explore potentially optimal schedules. Thus, the action space should not include actions which lead to suboptimal solutions. Heuristics can be used to reduce the solution space by using fixed rules for choosing part of the solution. Previous attempts at solving FJSPT problems have used heuristics, as summarized in Table 2-1. While other heuristics maybe leave out potentially optimal schedules in order to reduce the solution space, we can prove that $S_t = S_p$ will not.

Proof by induction. Consider an action sequence S_a consisting of a tuple (j, k, m) for each scheduled operation o_{ij}. A single operation schedule $|S_a| = 1$ where p_{ij} is scheduled immediately after t_{ij} is optimal, as the makespan is equal to the sum of p_{ij} and t_{ij}. Adding an operation to S_a so $|S_a| = n + 1$ leads to interactions between the current schedule and the added operation. Let us call an arbitrarily scheduled operation o_{sched} and a newly scheduled operation o_{new}.

Consider the interaction between o_{sched} and o_{new} in case $k_{\text{sched}} \neq k_{\text{new}}$, as t_{sched} and t_{new} happen on different vehicles they would be scheduled the same no matter which would be scheduled first as seen in Figure 4-3. Thus, using $S_t = S_p$ would not exclude any solutions. This holds for $k_{\text{sched}} \neq k_{\text{new}}$, $m_{\text{sched}} \neq m_{\text{new}}$ or $k_{\text{sched}} \neq k_{\text{new}} \wedge m_{\text{sched}} \neq m_{\text{new}}$.

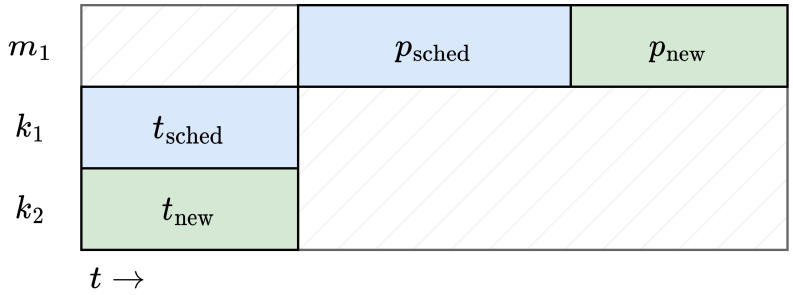

Figure 4-3: Schedule where $k_{\text{sched}} \neq k_{\text{new}} \wedge m_{\text{sched}} = m_{\text{new}}$

In case $k_{\text{sched}} = k_{\text{new}} \wedge m_{\text{sched}} = m_{\text{new}}$ consider the situation where the process order is not equal to the transport order as seen in Figure 4-4. Without loss of time we can swap t_{sched} and t_{new}, now following (4-4) p_{sched} and p_{new} can be shifted to the left or stay depending on c_m, thus leading to an equal or better makespan. As all interactions result in an equal or better makespan when using $S_t = S_p$, we can conclude that no optimal solutions are left out by using the same sequence.

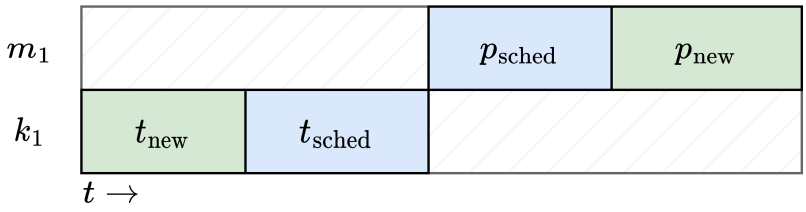

Figure 4-4: Schedule where $k_{\text{sched}} = k_{\text{new}} \wedge m_{\text{sched}} = m_{\text{new}}$

As all solutions were $S_t \neq S_p$ are eliminated, the solution space is reduced. As the order is predetermined, both the vehicle and machine can be chosen in a single action, leading to new action space 4-5.

$$\mathcal{A} = J_{\text{active}} \times K \times A \tag{4-5}$$

4-2 State representation

The state space S define the states of the MDP. This state representation is crucial as it has a big influence on learning performance [20]. The design of the state representation is not

trivial, as the state should contain enough information to comply with the Markov property but as well be fast to compute and use minimal memory.

4-2-1 State information

When solving a single problem instance, the minimal state information to comply with the Markov property is the (part) solution S_a as the problem instance would be static. However, a separate agent would be needed for every problem instance, so the problem instance P is included as well.

Further complexity is added by the fact that problem instances can differ in size. In case of the FJSPT, the size would be the number of machines, vehicles or operations. As the amount of information in P and subsequently S_a differs, the amount of information in the state is not static. This causes a problem as the policy function π is typically represented by an artificial neural network, which has a static input size. This can be accounted for in a number of ways,

1. A padding strategy where the state has a static size corresponding to the maximum problem instance size, in case of a smaller problem instance size, the state is padded with zeros [43].
2. A condensing strategy where the state information is condensed into a number of global criteria, such jobs completed, work left or average machine usage [11].
3. Using a graph based state encoding that is size invariant, such as Graph Neural Networks (GNN) [49]

A condensing strategy would invalidate the Markov property, resulting in limited learning performance. A padding strategy is limited to a maximum sample size, and its generalization properties are unknown [43]. However, GNN have been demonstrated to generalize over different problem instance sizes [42], [46], [50], making GNN the best choice.

4-2-2 Graph Neural Networks

GNN are a type of artificial neural network layer which can use graph data as input and output. While there are many types of GNNs [49] we will limit our scope to Convolutional Graph Neural Networks (ConvGNNs).

Using notation from Wu, Pan, Chen, *et al.* [49]: a graph is represented by $G = (V, E)$ where V is the set of nodes and E the set of edges. Let $v_i \in V$ denote a node and $e_{ij} = (v_i, v_j) \in E$ denote an edge from v_i to v_j. The neighbourhood of node v is defined as $N(v) = u \in V|(v, u) \in E$. A graph may have node attributes $X \in \mathbb{R}^{|V|\times d}$ for each node with $x_v \in \mathbb{R}^d$ representing the feature vector of node v. A graph may as well have edge attributes $X^e \in \mathbb{R}^{|E|\times c}$ with $x^c_{v,u}$ representing the feature vector of edge (v,u).

ConvGNN are similar to convolutional neural networks (CNN) [51] in that data is propagated by combining it with neighbouring data, however where in 2D CNN nodes are ordered and have a fixed number of neighbour in ConvGNNs they do not, as illustrated in Figure 4-5. A

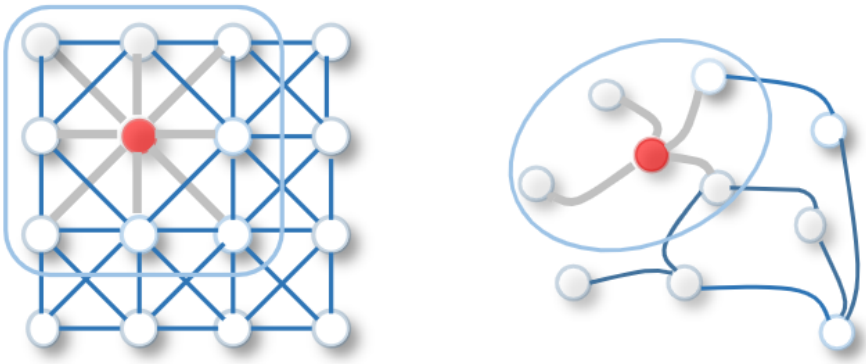

Figure 4-5: 2D Convolution vs. Graph Convolution of red node from [49]

commonly used ConvGNN layer is the Graph convolutional layer (GCNConv) [52]. Where the propagated node features $\mathbf{x}'_i$ are calculated as in (4-6).

$$\mathbf{x}'_i = \sum_{j \in N(i)} a_{ij} W \mathbf{x}_j \tag{4-6}$$

Where $W \in \mathbb{R}^{d \times d}$ is a learned matrix and a_{ij} is either a normalization factor from the node degrees $a_{ij} = 1/\sqrt{|N(i)||N(j)|}$ or in case edge attributes are used.

$$a_{ij} = \frac{x^c_{j,i}}{\sqrt{\sum_{j \in N(i)} x^c_{j,i}} \sqrt{\sum_{i \in N(j)} x^c_{i,j}}} \tag{4-7}$$

Graph Attention Networks (GATs) [53] instead learn the relation between node and edge attributed and the importance of the neighbour using attention networks. For example, in GATv2 [54].

$$\alpha_{ij} = \frac{1}{z_i} \exp(\mathbf{a}^T LeakyReLU(W_3 \mathbf{x}^c_{j,i} + W_2 \mathbf{x}_i + W_1 \mathbf{x}_j)) \tag{4-8}$$

Where $W_1 \in \mathbb{R}^{d \times d}$, $W_2 \in \mathbb{R}^{d \times d}$, $W_3 \in \mathbb{R}^{d \times c}$ and $a \in R^d$ are learned matrixes and z_i a normalization factor over all edges of i. When compared with other GNNs, GATv2 showed superior performance in node prediction datasets [54].

4-2-3 State design

To use the size agnostic properties of GNNs, the state of the FJSPT is represented as a graph. As discussed in Chap. 4-2-1, the state should contain both the problem instance and (part) solution. Disjunctive graphs have been demonstrated to represent this information for job shop scheduling problems [46], [55], flexible job shop scheduling problems [50], [56] and the FJSPT [13].

A disjunctive graph for a simple scheduling problem can be described by $\mathcal{G} = (\mathcal{O}, \mathcal{C}, \mathcal{D})$ where $\mathcal{O} = \{o_{ij} | \forall i, j\} \cup \{S, T\}$ are the nodes representing all operations plus a start and end dummy node. $\mathcal{C}$ the *conjunctive* edges which represent the order of the job operations. And $\mathcal{D}$ the

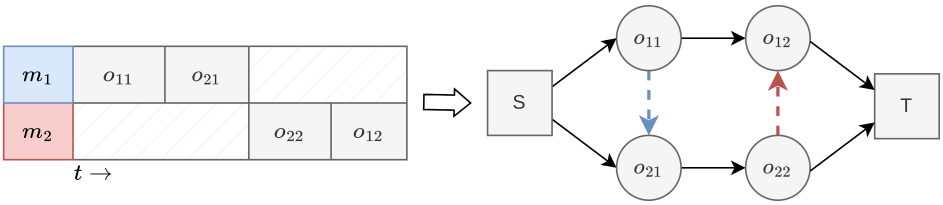

Figure 4-6: Disjunctive graph of schedule from a simple scheduling problem, disjunctive edge represented by dotted lines

disjunctive edges representing the order of operations using the same resource. In Figure 4-6 the disjunctive graph of a fully scheduled simple scheduling problem is given.

When an operation is not yet scheduled, undirected disjunctive edges are used to represent operations using the same resource. Each operation is connected to all other operations which can be run on the same resource. The amount of connections quickly scales with the number of operations, even more so when adding additional disjunctive edges for transport resources.

A possible solution is to only disjunctive edges for scheduled operations, as done in [55]. However, this invalidates the Markov property, as the machines available for unscheduled operations are no longer represented. In Song, Chen, Li, *et al.* [50] and Dax, Li, Leahy, *et al.* [56] additional nodes are added instead to represent machines as seen in Figure 4-7. By encoding S_p in node attributes X via the (expected) start time, the disjunctive edges $\mathcal{D}$ are no longer needed to represent the process sequence. A new type of edges, O-M edges $\mathcal{E}_{o,m}$ are introduces to represent machine assignment O_m. As a unique edge exists between every machine and operation combination, we can embed the process time as an edge attribute.

$$x^c_{o_{ij},m} = p^m_{ij} \tag{4-9}$$

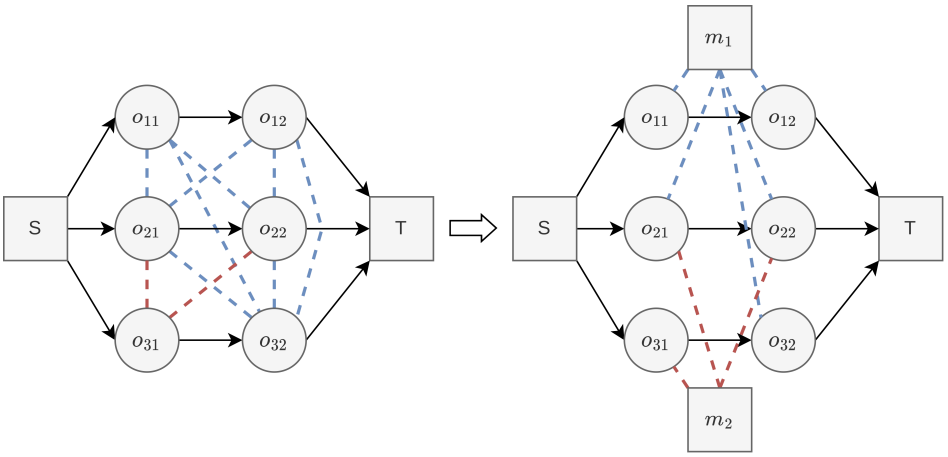

Figure 4-7: Replacing disjunctive edges $\mathcal{D}$, left, with O-M edges $\mathcal{E}_{o,m}$, right, $|J| = 3$, $|O| = 6$, $|M| = 2$

To use this representation for the FJSPT, we extend it by adding vehicle nodes. We introduce M-K edges $\mathcal{E}_{m,k}$ between vehicle- and machine nodes. As a unique edge exists between every

machine and vehicle, we embed the time to drive from the current location of vehicle to the machine.

$$x^c_{m,k} = t^{mm_k} \tag{4-10}$$

We introduce O-K edges $\mathcal{E}_{o,k}$ between operation- and vehicle nodes, here we embed the average setup time for the vehicle to drive to the operation. Where $A_{ij} = O_m(o_{ij})$ once a machine has been assigned, and all machines capable of processing o_ij before.

$$x^c_{o_{ij},k} = \frac{\sum_{m \in A_{ij}} t^{m_k m}}{|A_{ij}|} \tag{4-11}$$

This disjunctive graph can be described by $\mathcal{G} = (\mathcal{O}, M, K, \mathcal{C}, \mathcal{E}_{o,m}, \mathcal{E}_{m,k}, \mathcal{E}_{o,k})$, an example is given in Figure 4-8. A summary of how the state information is encoded is given in Table 4-1.

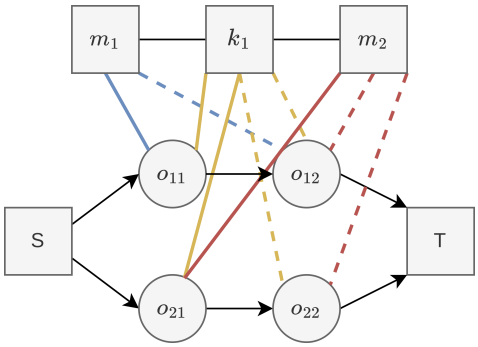

Figure 4-8: FJSPT disjunctive graph $\mathcal{G}$, unscheduled edges are dotted, with $|J| = 2$, $|O| = 4$, $|M| = 2$, $|K| = 1$

Table 4-1: State components and their encoding

Problem instance		*(Part) solution*		
$\boldsymbol{p^m_{ij}}$	$\boldsymbol{t^m_{ij}}$	$\boldsymbol{O_m}$	$\boldsymbol{O_k}$	$\boldsymbol{S_p}$
$x^c_{v,u} \forall (v,u) \in \mathcal{E}_{o,m}$	$x^c_{v,u} \forall (v,u) \in \mathcal{E}_{m,k}, \mathcal{E}_{o,k}$	$\mathcal{E}_{o,m}$	$\mathcal{E}_{o,k}$	$x_v \forall v \in \mathcal{O}$

Additionally, we add some precomputed features to X, as these are quick to calculate the learning speed-up will likely outweigh the calculation penalty. We embed the following features in X, one hot encoding of the node type: $\mathcal{O}$, M or K. Boolean if an operation can be scheduled next. Boolean if all operation is scheduled, or all operations are done on a resource. The earliest possible start time of the operation or resource. The number of operations left per job per operation or per resource. The (average) duration over all available machines per operation or for all transport and process tasks left per vehicle and machine, respectively. The number of connections to other nodes. The definition of all node attributes, and their respective normalization factors, are given in Appendix A.

4-2-4 Network design

The total network takes as input the current state of the system represented by $\mathcal{G} = (\mathcal{O}, M, K, \mathcal{C}, \mathcal{E}_{o,m}, \mathcal{E}_{m,k}, \mathcal{E}_{o,k})$ and outputs a policy $\pi(s)$ and value $V(s)$. The total struc-

ture of the network is depicted in Figure 4-9. The function of all parts is described below, the exact architecture of each component is described in Appendix B.

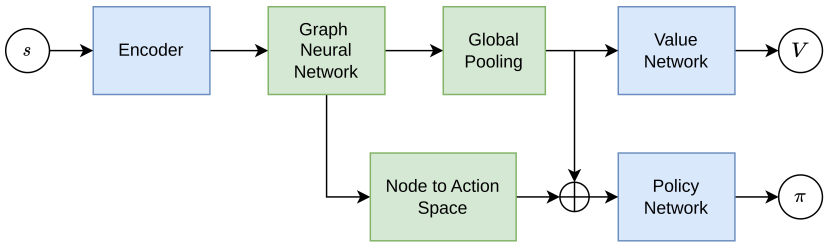

Figure 4-9: Network architecture, node attribute based operations in blue, graph based operations in green

The **encoder** encodes the raw node attributes before the graph is run through the graph neural network. This is done to first encode the boolean node attributes, such as node type, as otherwise the GNN summation (4-6) would sum raw node attributes of different node types.

The **GNN** encodes the graph properties in the node attributes. To deal with the different types of nodes, a GATV2 structure is used. As attention heads are used, different types of nodes can be weighted differently as the type is encoded in the node attributes.

The **global pooling** pools the node attributes for all types of nodes, representing the global state for all operation, machine, and vehicle nodes.

The **node to action space** concatenates the node attributes of active operations, machines and vehicles to form the action space $\mathcal{A}$. For each action, the attributes of the corresponding operation, the vehicle responsible for transport and the machine for processing the operation are concatenated to form the action attributes. Additionally, the global attributes are concatenated.

The **value network** determines the value of the state based on the global features. The **policy network** determines the policy of the state based on the action attributes.

4-3 Rewards

The reward function $R(s)$ defines the reward awarded for reaching state s. In order to train the agent towards finding the best solution, the reward is defined such that maximizing the reward will result in finding the best solution.

4-3-1 Reward definition

The reward is defined as minus the makespan for all states that are terminal. As all other rewards are 0, the discounted sum of rewards G_t, where $\gamma = 1$, equals $-c_{\max}$.

$$R(s) = \begin{cases} -c_{\max}, & |S_p| = |O| \\ 0, & \text{otherwise} \end{cases} \tag{4-12}$$

This results in a sparse reward function, this is however not a problem as the MCTS either expands to a terminal node or estimates V(s).

4-3-2 Normalization

To maximize the sensitivity of the reward, ideally the best solution is giving reward 1, while the worst reward is given reward 0. However, the best and worst reward are not known apriori.

A simple method is to use a heuristic to calculate the upper and lower bound, UB and LB, of the problem instance. These can then be used to normalize the reward to form the bounded reward.

$$R(s) = \begin{cases} r, & |S_p| = |O| \\ 0, & \text{otherwise} \end{cases}, \quad r = \frac{\text{UB} - c_{\max}}{\text{UB} - \text{LB}} \tag{4-13}$$

Calculating sufficiently accurate upper- and lower bounds is however difficult in practise, as these bounds need to be very tight to offer sufficient range over V and needs to be consistent over problem sizes.

Adaptive normalization uses the information found during search for normalization instead. A number of methods have been used to achieve this.

State based normalization normalizes all future rewards on a per state basis. By normalizing all rewards in state s with all other rewards found from state s the rewards are normalized. In [57], [58] this was realized normalizing the rewards using the mean and standard deviation of all future rewards from state s, μ_s and ϵ_s, $r = \frac{-c_{\max} - \mu_s}{\epsilon_s}$ While in [59] the rewards were normalized by the best and worst rewards found in each state, $r = \frac{R^s_{\min} - c_{\max}}{R^s_{\max} - R^s_{\min}}$. Conceptionally there is a problem however, as the rewards are normalized per state but back propagated to parent states. During search, the reward of the best child node of a bad part solution would be back propagated with the maximum reward, steering towards a bad solution.

Global normalization [60] uses a moving average instead to determine μ and ϵ over all rewards. In practise, the average reward for combinatorial problems however shifts depending on the number of scheduled operations. This causes the moving average to drift, causing an inconsistent value estimate.

In order to use a global normalization method without drift, the rewards are normalized using the best and worst rewards found during search over all states.

$$R(s) = \begin{cases} r, & |S_p| = |O| \\ 0, & \text{otherwise} \end{cases}, \quad r = \frac{R_{\min} - c_{\max}}{R_{\max} - R_{\min}} \tag{4-14}$$

This way the rewards are normalized consistent over a problem instance and the sensitivity is maximized over the found rewards. Like the ranked rewards [61], [62] method, where the top percentile of rewards found are trained as the maximum reward while the others are given the minimum reward, the scale shifts if the agent improves and is able to find better rewards. Thus, the value function keeps sensitivity even if the agent improves.

4-3-3 Leaf inclusion

To train the value function to differentiate between good and bad solutions, the training data should include samples from both types. While in a two player AZ game the whole reward space is explored naturally, as both players have equal chance to win, in a single player setting this is not the case. As only the samples from the solution are added to the training data, only high value samples are added. As a result $V_{\boldsymbol{\theta}}$ will bias to a high value estimation for all samples. To remedy this, we introduce leaf inclusion, a novel method to generate a more uniform value spread.

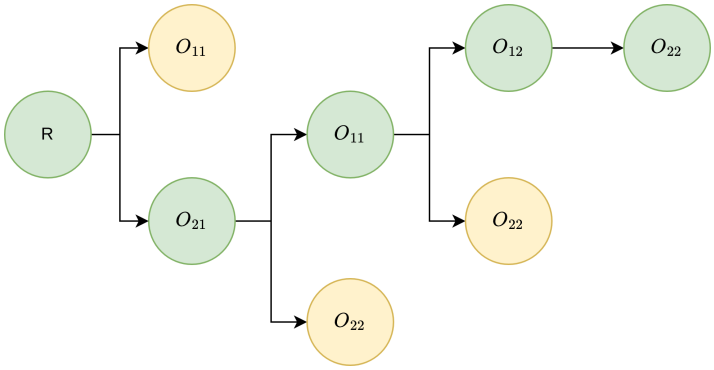

Figure 4-10: MCTS tree, leaf inclusion includes the yellow nodes

Going by the idea that visiting a node more often improves the value estimate, leaf inclusion includes first level children of the solution to the training buffer. The value of the solution nodes are always 1.0 as it the best solution found, the values of the leaf nodes are determined by the average reward received.

$$V_{\text{mcts}}(s_t) = \begin{cases} 1.0, & s_t \in S_p \\ \frac{\sum_{i=1}^{n(s_t,a_t)} R_i(s_t,a_t)}{n(s_t,a_t)}, & \text{otherwise} \end{cases} \tag{4-15}$$

As each node in the solution is used as root node during search, the children nodes are visited more often than other nodes. However, the number of visits varies greatly per node and consequently the accuracy of value estimate and policy. To remedy this, the samples are weighted by weight w based on the number of visits. The weight formula is based on the confidence bound of the UCT formula (3-12) as samples with a smaller confidence bound should be weighted more.

$$w_t = \sqrt{\frac{n(s_t, a_t)}{\ln(\sum_a n(s_t, a_t))}}$$

The loss function adjusted to include the weight.

$$L_\pi(\boldsymbol{\theta}) = -\sum_t w_t \pi_{mcts}(s_t) \log\left(\pi_{\boldsymbol{\theta}}(s_t)\right) / \sum_t w_t$$

In the value loss function, the weight is squared as fewer samples are needed to estimate the value. Additionally, to increase sensitivity of the value function, the loss is multiplied by a

factor 100.

$$L_V(\boldsymbol{\theta}) = 100 \sum_t \sqrt{w_t} \Big(V_{\boldsymbol{\theta}}(s_t) - V_{mcts}(s_t) \Big)^2 / \sum_t \sqrt{w_t}$$

To compare the effectiveness of the leaf inclusion, the value of the samples in the training data is compared to the ones used in inference. The inference samples were determined by binning the values of all nodes encountered in 10 random problem instances, while the training data is sampled from the training buffer and weighted by w.

The value spread is compared between an agent trained using a bounded normalization from (4-13) and an adaptive strategy using (4-15) with leaf inclusion. Comparing Figure 4-11 and 4-12 it can be observed how the training data covers a wider range, allowing the value function to differentiate between good and bad solutions better. And secondly, increased overlap with the inference data.

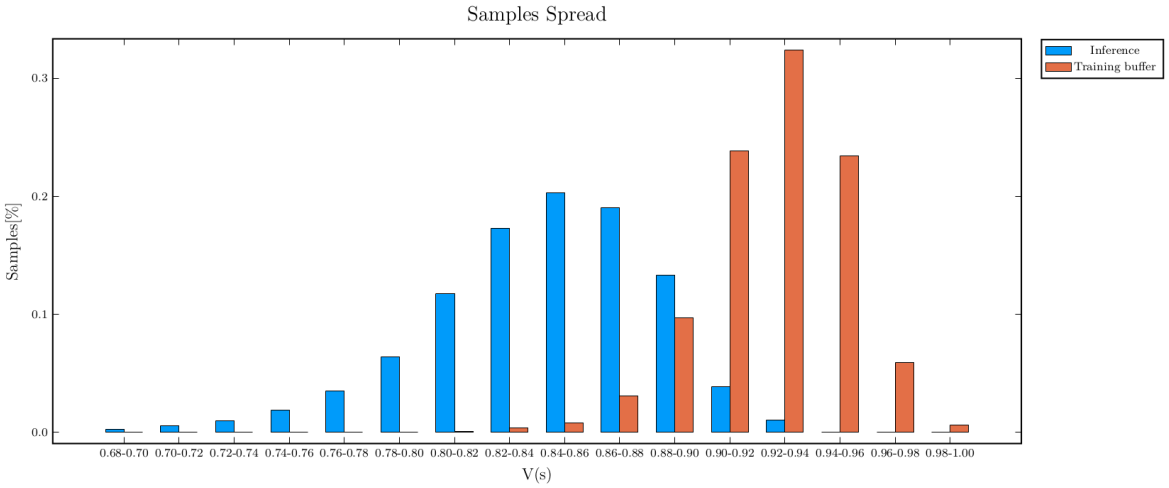

Figure 4-11: Inference versus training data spread for bounded normalization

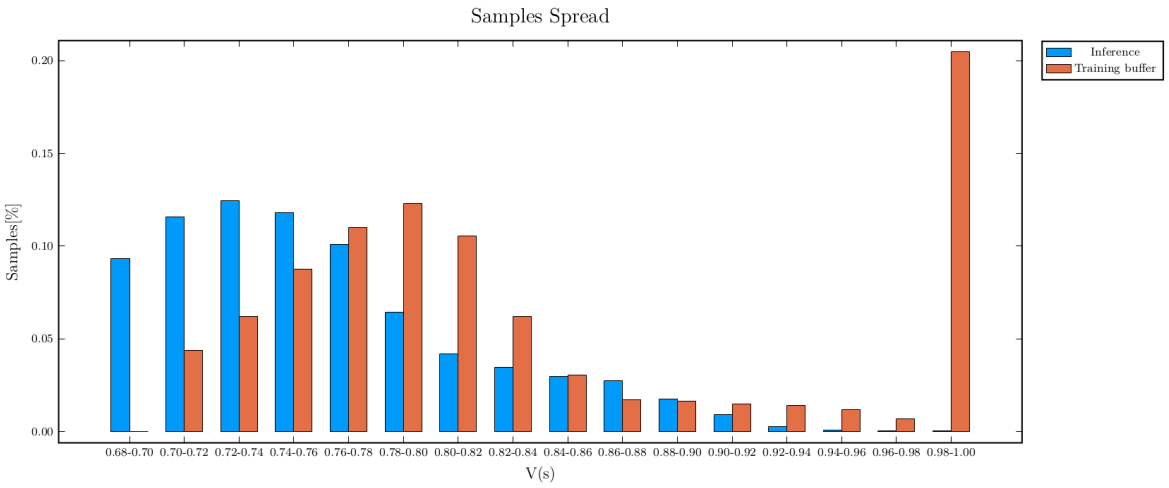

Figure 4-12: Inference versus weighted training data spread with leaf inclusion

An MDP implementation was designed for the FJSPT. A reduced action space was constructed and a size agnostic state representation based on GNN. The complexities of reward normalization were discussed, and a novel method called leaf inclusion is introduced. Leaf inclusion successfully allows for a more diverse value function training data set, the implementation and results of this design are discussed in the next chapter.

Chapter 5

Implementation & Results

Following the design of the algorithm, the details of the implementation are discussed and the results found.

5-1 Implementation details

To implement the combinatorial AlphaZero algorithm software and parameter design choices were made, this chapter details the choices for the implementation.

5-1-1 Software implementation

To implement the software, an existing AlphaZero implementation was extended to allow for solving combinatorial problems. Several additions were made to make this possible.

Julia language

To implement the algorithm, the Julia language [63] was chosen over the common choice of Python for multiple reasons.

- Just in time (JIT) compilation allows for near C execution speed, allowing all code to be implemented in a single language, where Python implementations need MCTS implemented in C externally
- Excellent machine learning support by the Flux.jl [64] package
- Native support for distributed and multi-gpu computing in Distributed.jl
- Implementation of the AlphaZero algorithm in AlphaZero.jl [65]
- Implementation of GNN in GraphNeuralNetworks.jl [66]

Graph batching

The state representation in AlphaZero.jl was modified from matrix based representation to graph objects. While in a game setting there is a fixed state and policy size, in a combinatorial setting both are dependent on the problem instance size. In a game setting, a masking strategy is used to eliminate moves that become invalid during the game, as combinatorial problems can have a large range of number of moves possible a padding strategy was implemented instead. As the training data is adaptively padded to the largest instance in the training data, problem instances can be mixed in the training data.

Adaptive normalization

Adaptive normalization was implemented as specified in chapter 4-3. Each problem instance records the best and worst rewards found so far, $R_{\min}, R_{\max}$. While in a vanilla AZ implementation, each MCTS node holds the sum of rewards W and number of visits N where the average reward $\bar{R} = W_i/N_i$, adaptive normalization complicates this. As $R_{\min}, R_{\max}$ are constantly updated, the unnormalized values are stored and are normalized at evaluation time. As the estimates from $V_{\boldsymbol{\theta}}$ are normalized, the unnormalized values are stored separate as W^t, N^t. The node info is updated as:

$$\text{Node}(s,a)_{i+1} : \begin{cases} W^t_{i+1} = W^t_i + \frac{R_{\min} - R_i(s,a)}{R_{\max} - R_{\min}}, & N^t_{i+1} = N^t_i + 1, R_i \text{ is terminal} \\ W_{i+1} = W_i + R_i(s,a), & N_{i+1} = N_i + 1, \text{ otherwise} \end{cases}$$

Where the average rewards is

$$\bar{R} = \frac{W_i/N_i + W^t_i/N^t_i}{2}$$

Graph neural networks

Graph neural networks were added using GraphNeuralNetworks.jl for the graph wide operations and Flux.jl for the node attribute operations. To speed up inference and training, custom batching and unbatching functions were developed to only encode necessary information. To prevent unnecessary transfers between the CPU and GPU, the state evaluation was implemented fully on the GPU, including the node to action space process as defined in Figure 4-9.

FJSPT implementation

The FJSPT problem was implemented with a focus on keeping the memory print of each state minimal. As the search tree contains every state it has encountered, keeping the memory size small is important for performance. The internal state consists of three parts, the problem instance, the solution and the current state. The problem instance is stored per problem instance as it is static, while the solution and state are updated and stored per state. The state composition is summarized in Table A-1. To estimate the state policy and value, each state is converted to a graph which is inputted to the GNN.

Table 5-1: State data

Type	Data	Size
Problem instance	Number of machines	1
	Number of jobs	1
	Number of vehicles	1
	Number of operations per job	N
	Process time per operation, machine	$O \times M$
	Transport time from, to	$M+1 \times M+1$
Solution	Assigned vehicle, machine per operation	$O \times 2$
	Ready time, transport, process	$O \times 2$
	Done time, transport, process	$O \times 2$
State	Last operation on machine	M
	Last operation on vehicle	K

5-1-2 Training data

A condition for improving the agent is that the MCTS is able to find a better policy π_{mcts} than the current policy π as supervised/imitation learning is used to improve the policy. When problem instances become too large and the agent is not yet trained, the MCTS might not be able to find good solutions yet as the value and policy are pointing in random directions. To prevent this, an appropriate sized problem instance should be used for training. To verify this, for increasingly large problem instances, the MCTS solutions are compared with greedily following the current policy π. In Figure 5-1 the average solution quality, calculated by the bounded equation (4-13), is compared for jobs with $N, M, A = 1$ until $N, M, A = 6$. It can be observed that even if the problem size increases, the MCTS is able to improve over the current policy. Verifying the policy improvement capabilities of the MCTS over a wide range of problem instance sizes.

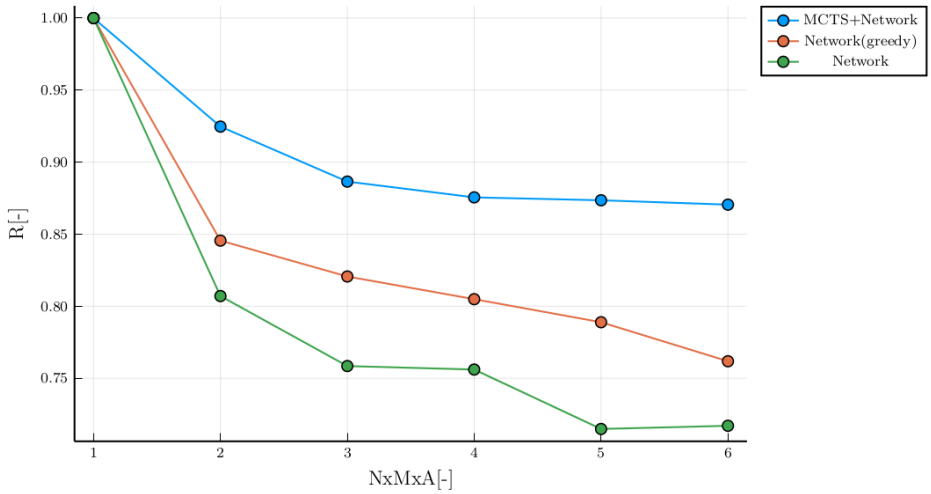

Figure 5-1: Normalized reward for problem instances with increasing size, MCTS with 500 iterations per step

To train for various sized problems, a mixed problem size training set is used. As the AlphaZero implementation was designed for allowing mixed sized instances, as explained in section 5-1-1, learning happens over a mixed set of problem sizes. The size of the training set was based on the benchmark instances. All training size parameters are in Table 5-2.

Table 5-2: Size of problem instances training set

	M	N	A	K	P	T
Min	4	3	1	2	1	1
Max	8	8	3	2	128	128

5-1-3 Exploration

To promote exploration, as per the AlphaZero paper [31], Dirichlet noise = Dir(α) can be added to the policy of the root node. By adding noise to the root node, further exploration is promoted to prevent the agent getting stuck in local minima. Two hyperparameters are introduces by this process α and ϵ augmenting the root nodes policy π_{mcts} as

$$\pi^*_{\text{mcts}}(s) = (1-\epsilon)\pi_{\text{mcts}}(s) + \epsilon \text{Dir}(\alpha)$$

Furthermore a temperature hyperparameter can be used to, instead of greedily following the MCTS policy, pick from the policy distribution. However, in a combinatorial optimization setting, it does not make sense to schedule a suboptimal action. In practise, the temperature hyperparameter was found to be detrimental to performance.

Lastly, the exploration parameter c is used to balance exploration and exploitation in the UCT formula. The following hyperparameters were tuned by hand for optimal results. $c = 0.8$, $\alpha = 0.15$, $\epsilon = 0.1$.

5-1-4 Learning

A limitation of the AlphaZero.jl package is that iterative learning is used. In iterative learning, self-play and learning steps are alternatively performed. In the self-play step, new samples are collected by solving problems and added to the learning buffer, after which in the learning step the policy and value networks are updated by learning over the buffer. A downside of this implementation is that the policy and value functions are only updated once every iteration.

While in the original AlphaGo paper [33] iterative learning was used, in the AlphaZero paper [31] continues learning was used. In continues learning, the learning is performed in parallel to the self-play. As the policy and value networks are constantly updated, the learning is stabilized [31]. However, this mechanism is hard to implement in software, causing open access AlphaZero implementations to choose for iterative learning.

A gradient descent optimizer with Nesterov acceleration was used to optimize during the learning step. In practise, this optimizer was the most stable optimizer. The learn-rate was observed to be of great importance in learning speed and stability. In Figure 5-2 the difference in loss L after a learning episode with different learnrates is shown. It can be observed that a negative ΔL is only acquired with a range of learnrates, a smaller or larger learnrate will lead to regression. To stabilize learning, the learnrate was tuned regularly.

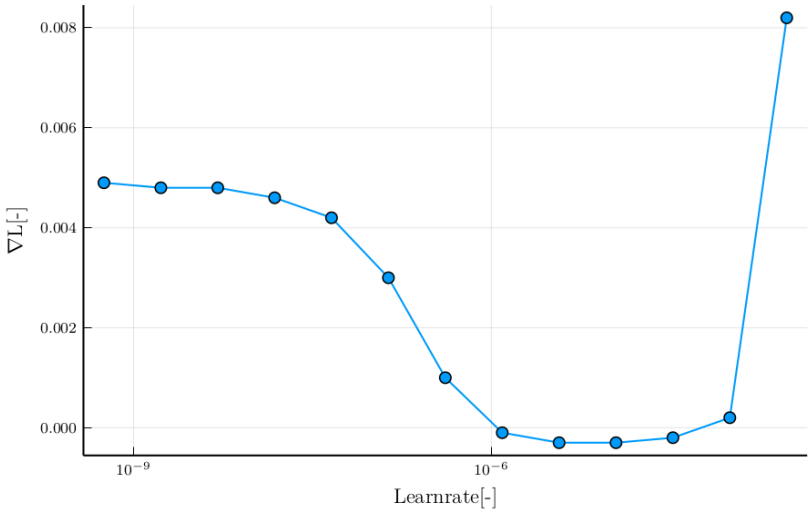

Figure 5-2: Loss delta versus learnrate

5-2 Results

5-2-1 Benchmark set

To determine the performance of the algorithm, a benchmark set is used. An extensive FJSPT benchmark set was compiled by Homayouni and Fontes [7], combining benchmark sets used in previous work as well as introducing new large sized sets. As the set consists of small, medium, and large sized instances, a wide range of problem instances can be benchmarked. A subset of the benchmark set, with various sizes, was picked to keep the results manageable.

5-2-2 Benchmark results

In Figure 5-3 the results of the benchmark set are plotted for a number of solvers.

- The late acceptance hill climbing algorithm from Homayouni and Fontes [7], currently the best known makespan
- The MCTS plus GNN implementation performing 3000 iterations per step
- A greedy implementation of just the GNN
- An heuristic implementation where the shortest process sequence (SPS) is used for job selection and shortest process time (SPT) is used for assigning the machine and vehicle

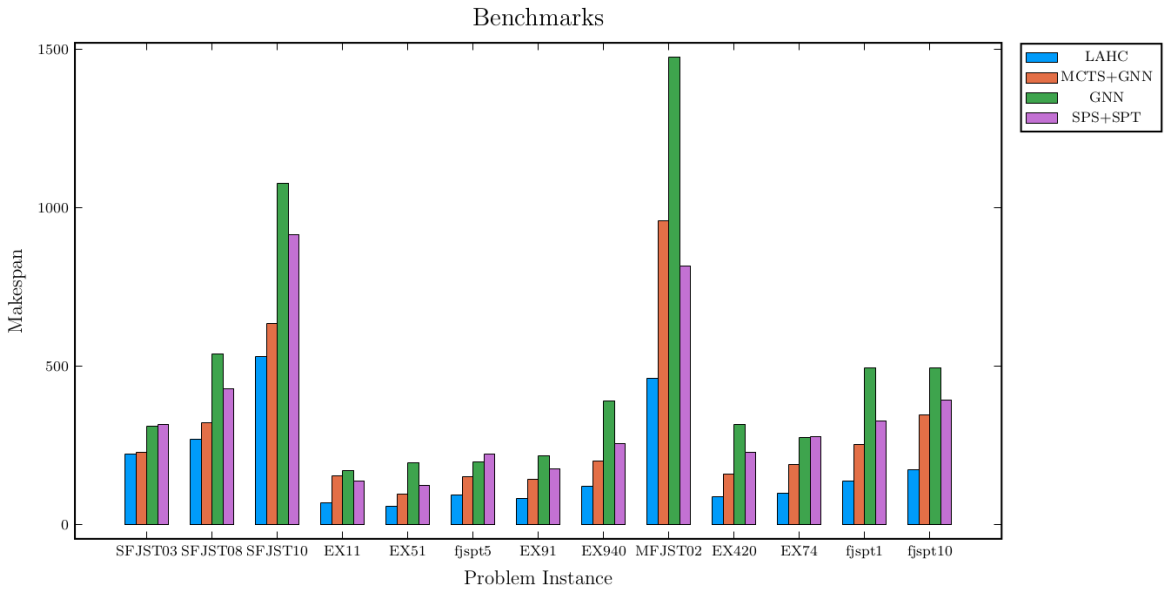

Figure 5-3: Best makespan found by algorithms for problem instances in benchmark set

While the algorithm is able to find better solutions than the heuristic for all but one instance, the makespan found for all but SFJSPT03 are nowhere near optimal. As the results are not competitive, the rest of this thesis will focus in determining the cause.

5-2-3 Process overview

To determine the cause of the agent not finding optimal solutions after training, we decompose the learning process of the AlphaZero framework. As displayed in Figure 5-4 the process consists of a self-play and learning step interacting. When working correctly, the self-play step uses the current policy $\pi_{\boldsymbol{\theta}}(s,a)$ and value function $V_{\boldsymbol{\theta}}(s)$ to find an improved policy $\pi_{\text{mcts}}(s,a)$ and value function $V_{\text{mcts}}(s)$ using MCTS. Subsequently, $\pi_{\text{mcts}}(s,a)$ and $V_{\text{mcts}}(s)$ are used to update $\pi_{\boldsymbol{\theta}}(s,a)$ and $V_{\text{mcts}}(s)$ in the learning step in a supervised learning setting.

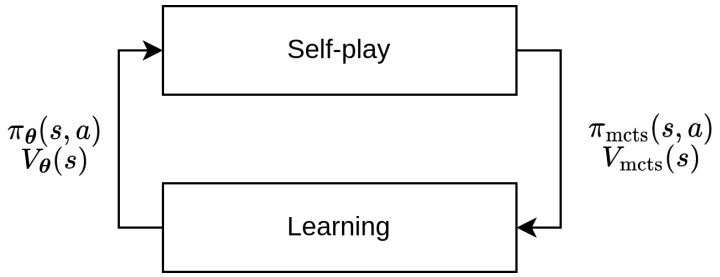

Figure 5-4: High level overview of the AlphaZero algorithm learning process

To determine the problem, we verify each part of the process.

5-2-4 Self-play

The goal of self-play is to improve the current policy and value function, both functions are evaluated.

Policy improvement

As the optimal policy maximizes the reward, a policy is improved if it lowers the makespan. Thus, to determine if a policy is improved by the MCTS, we compare if the MCTS policy $\pi_{\text{mcts}}(s, a)$ is able to find better solution than the current policy $\pi_{\boldsymbol{\theta}}(s, a)$.

MCTS is an anytime algorithm that approaches optimality when run for infinite iterations [34], increasing the number of iterations increases the search space and theoretically the solution quality. To determine if the policy is improved by the MCTS, the average reward when following $\pi_{\boldsymbol{\theta}}(s, a)$ and $\pi_{\text{mcts}}(s, a)$ after a number of MCTS iterations, was compared for 30 problem instances. The results can be seen in Figure 5-5. After 1200 iterations per step, all policies were improved. As 1500 iterations per step were used during training, the policy improvement works correctly.

Figure 5-5: Percentage of policies improved after number of MCTS iterations

Value function improvement

While the MCTS search is directed by the policy, the value function is important in steering the search in the initial number of steps. As for deep search trees, initially no terminal nodes are reached during the search relies on the value function for finding promising regions. As bad initial choses can lead to poor solutions, a well performing value function is required for finding good solutions.

Due to the calculation of $V_{\text{mcts}}(s)$ in (4-15), the value of nodes are equal to the average reward reached from the state. States close to terminal states can form an accurate estimate of $V(s)$

without a trained $V_{\boldsymbol{\theta}}(s)$ as terminal states return the true reward. As the network is trained and these states are estimated well, states close to these states can form an accurate estimate of $V(s)$ creating a cascading effect.

When plotting the loss over the number of unscheduled operations in a state in Figure 5-6, it can be observed that this cascading effect had not progressed. The states closer to terminal states, which are estimated better as the MCTS reaches terminal states more often, have a larger loss than the states farther away. This can be explained by the fact that the leaf nodes of the states farther away from the terminal states are calculated by the average reward reached, as this reward is an estimated reward from the value function the estimate is trapped in a local minimum as it is not affected by the true value of a state.

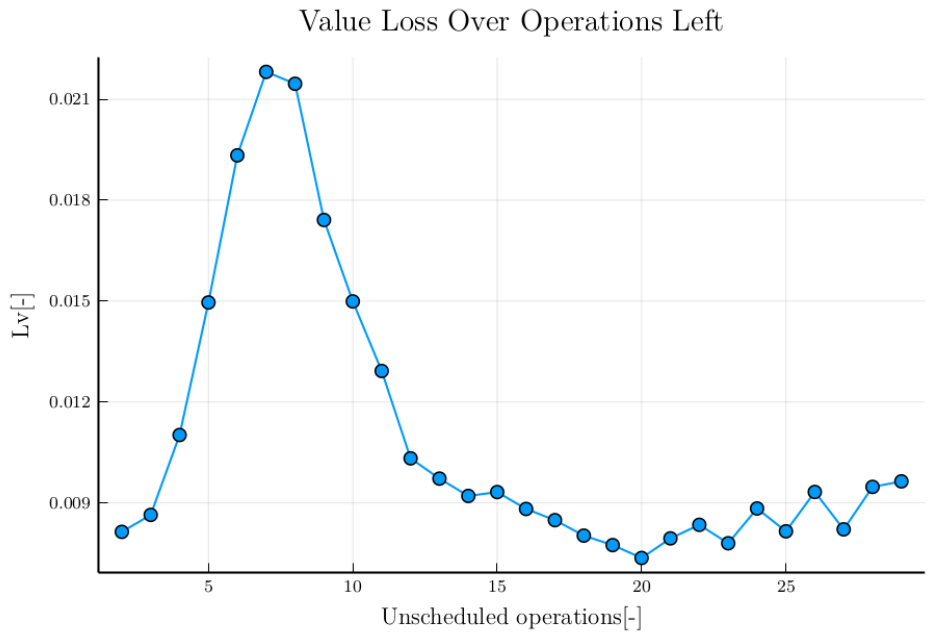

Figure 5-6: Value loss L_v over the number of operations left

A way to prevent this would be to weight the value samples by the number of terminal node visits N^t instead of the total visits $N + N^t$. However, this would lead to undersampling of the early leaf nodes. A correct value function is crucial for the early stages of the search, however correctly fitting it is a complicated problem for combinatorial optimization.

5-2-5 Learning

The learning step trains the GNN to output a better policy and value estimate.

Network architecture

To test if the GNN architecture is able to work as an effective policy and value function, the agent was trained on small problem instances. As smaller problem instances have limited

depth, the MCTS is able to find good policies consistently, eliminating other variables that might prevent learning. After training on $N, M, A = 3$ the policy improved to an average bounded reward of 0.88 and the AZ agent is able to find optimal solutions for small instances. This indicates the GNN is able to work effectively as policy for small instances.

When training on larger instances, the policy improves slowly as the loss decreases. However, samples are gathered significantly slower during self-play as larger solution spaces need to be explored. Another limiting factor could be the size of the GNN, a larger network can encode more complex connections between input and output data. The current network contains 310,706 parameters. As the network is evaluated for every node, increasing the network size further would significantly slow down inference, decreasing the amount of training data. Thus, the network size is limited by the speed of the implementation.

As training-speed and efficiency influences how well the networks can be trained, the next section will discuss problems with the current implementation.

5-2-6 Training efficiency

The AlphaZero is a resource intensive algorithm, as it requires powerful CPU resources for the MCTS search and GPU resources for the neural network evaluations and training. Many optimizations can be made in this process, as all components need to work together efficiently.

The implementation in this thesis was built on top of the AlphaZero.jl package. While the implementation uses batching techniques to evaluate states in batches to more efficiently use the GPU, the average GPU usage was limited to 20%. The bottleneck of the implementation was caused by the transfer speed between the CPU- and GPU memory. A lot of effort was put into diminishing this problem, however this was ultimate out of scope for a thesis. Additionally, the GraphNeuralNetworks.jl package is still actively developed as of this thesis and misses crucial optimizations, as sparse GPU matrix multiplication is not fully implemented.

The efficiency of the AZ algorithm for scheduling versus other reinforcement learning methods is also doubted, as Wang, Cheng Luo, Xiong, *et al.* [43] found PPO far more efficient in training a scheduling policy than an MCTS based approach. Although this is very much dependent on the speed of the implementation.

A result of the poor training speed is the need for very long experimentation cycles, as multi day training sessions are needed to evaluate a method. Combined with the complexity of the FJSPT problem, the black box nature of the (graph) neural networks and the difficulties in training a correct value function makes the problem unsuitable for further thesis work. Further work should be focussed on developing an efficient AZ implementation for simple combinatorial problems, with focus on developing a functional value function. Afterwards, methods for representing the state can be evaluated, either as GNN or the more recently popular transformer [67].

Chapter 6

Conclusions and Outlook

The goal of this thesis was to develop a method for quickly calculating FMS schedules. The FJSPT model was chosen to model the complexities of an FMS system, including the transportation of materials. A systematic search method was chosen to find partial solutions quickly that can be used in a rolling horizon implementation in future work. No competitive systematic methods were developed yet for the FJSPT problem, so a new method was developed.

As the search space of the FJSPT is large, an efficient search algorithm was needed. Inspired by the results of AlphaZero in the complicated game of Go, the AlphaZero algorithm was chosen. The AlphaZero algorithm uses a reinforcement learning based method for improving its search iteratively.

An implementation for FJSPT in a reinforcement learning framework was made. In order to be problem instance size agnostic, graph neural networks were used. In order to limit the number of nodes in the network, a node combination strategy was used to go from node to action space. Adaptive normalization was used to create a sensitive value function, while leaf inclusion made sure the training data was diverse.

The method performs poorly in comparison to other optimization methods for the FJSPT. The value function is important in guiding the MCTS in the first number of steps, as no terminal nodes are reached. However, implementing a value function that is sensitive to solution quality, normalized and trained on a diverse set of values is difficult. Future work should be focussed on forming such a value function, as it is critical for choosing the best initial steps for large combinatorial problems.

Additionally, the AlphaZero algorithm is difficult to implement efficiently, as it makes use of both GPU and CPU resources. Inefficiencies cause long experimentation cycles, making research difficult. Due to the black box nature of (graph) neural networks and search algorithms, this makes the AlphaZero algorithm unpractical for thesis work. Combined with the complexity of the FJSPT problem, further analysis is difficult due to the many variables. Future work is advised to first develop an implementation for simple problems before analysing the FJSPT.

While no efficient search method was developed, the thesis presents a motivated MDP implementation for the FJSPT, a normalization method and insight into the complexities of AlphaZero for combinatorial optimization.

Appendix A

Node features

For all unscheduled operations, we propagate the completion time via the minimum completion time ignoring capacity constraints.

$$c_{t_{ij}} = c_{p_{ij-1}} + \min_{m \in A_{ij}} t_{m'm}$$

$$c_{p_{ij}} = c_{t_{ij}} + \min_{m \in A_{ij}} p_{ij}^{m}$$

Table A-1: State features X, in order to generalize over different sized problem instances, the features are normalized by $\mathcal{N}$

$x_{1...9}$	$\boldsymbol{\mathcal{O}}$	$\boldsymbol{M}$	$\boldsymbol{K}$	$\boldsymbol{\mathcal{N}}$
is_operation	1	0	0	1
is_machine	0	1	0	1
is_vehicle	0	0	1	1
is_next_operation	$\max_j o_{ij} \notin S_p$	0	0	1
is_done	$o_{ij} \notin S_p$	$m \notin A_{ij} \forall (i,j) \in O$	0	1
ready_time	$\max(c_{p_{ij-1}}, c_k + t^{m_k m'})$	c_m	c_k	$\max c_{p_{ij}}$
operations_left	$\sum_j (o_{ij} \notin S_p) \forall i \in J$	$\sum_{ij \in O} m \in A_{ij}$	$\sum_{ij \in O} A_{ij}$	$\sum_{ij \in O} A_{ij}$
duration	$c_{p_{ij}} - c_{p_{ij-1}}$	$\sum p_{ij}^m$	$\frac{\sum_{m \in M} t^{m_k m}}{\lvert M \rvert} + \frac{\sum_{m_1 \in M} \sum_{m_2 \in M} t^{m_1 m_2}}{\lvert M \rvert^2}$	$\max c_{p_{ij}}$
connections	A_{ij}	$\sum_{ij \in O} m \in A_{ij}$	$\sum_{ij \in O} A_{ij}$	$\sum_{ij \in O} A_{ij}$

Appendix B

Network details

B-1 Normalization

While in AZ [33] batch normalization [68] is used, our implementation doesn't. As batch normalization is an operation over all nodes, the operation quickly becomes infeasible due to VRAM limitations. Instead, the selu activation function [69] is used due to its self-regulating and gradient vanishing prevention properties.

B-2 Encoder

The encoder encodes the raw node data before it is fed to the GNN. Without an encoder, data could be lost in the GNN as nodes are combined. The encoder is a 3 layer dense MLP network with a hidden layer size of 48 with a selu activation function.

B-3 Graph Neural Network

The GNN consists of 3 blocks containing a three-headed GAT layer plus 3 layer dense network to combine the output of the GATv2 layer. Three heads were chosen to focus on each of the node types.

B-4 Global Pooling

Global pooling combines all node data to determine the global state $\mathbf{u}$ of the graph. The global state consists of the sum of the state of each node type, the next operation nodes $\mathbf{u}_O$,

the machine nodes $\mathbf{u}_M$ and the vehicle nodes $\mathbf{u}_K$.

$$\begin{aligned}
\mathbf{u}_O &= \sum_{i \in V \in O_{ij+1}} \mathbf{x}_i \\
\mathbf{u}_M &= \sum_{i \in V \in M} \mathbf{x}_i \\
\mathbf{u}_K &= \sum_{i \in V \in K} \mathbf{x}_i \\
\mathbf{u} &= [\mathbf{u}_O; \mathbf{u}_M; \mathbf{u}_K]
\end{aligned}$$

B-5 Node to action space

The node to action space combines the output data from the GNN to form a vector for each action possible. The feature vector for an operation, machine, vehicle combination $\mathbf{x}_{omk}$ consists of the features of the corresponding operation machine and vehicle plus the global data.

$$\begin{aligned}
A_{omk} &= J_{\text{active}} \times A \times K \\
\mathbf{x}_{omk} &= [\mathbf{x}_o; \mathbf{x}_m; \mathbf{x}_k; \mathbf{u}] \\
X_A &= [\mathbf{x}_{omk} \quad \forall o, m, k \in A_{omk}]
\end{aligned}$$

B-6 Value network

The value network estimates $V(s)$ from $\mathbf{u}$, the MLP network is a 5 layers deep with a hidden layer size of 96. Selu is used for the hidden layers, while the output uses a sigmoid function.

B-7 Policy network

The policy network estimates $\pi(s)$ from X_A, it uses 7 hidden layers with a width of 96 and selu activation function. The output is softmaxed over all actions of the graph.

Bibliography

[1] S. Mayer, T. Classen, and C. Endisch, "Modular production control using deep reinforcement learning: proximal policy optimization," *Journal of Intelligent Manufacturing*, pp. 1–17, May 2021, ISSN: 15728145. DOI: 10.1007/s10845-021-01778-z. [Online]. Available: https://link.springer.com/article/10.1007/s10845-021-01778-z.

[2] H. E. Nouri, O. B. Driss, and K. Ghédira, "Hybrid metaheuristics for scheduling of machines and transport robots in job shop environment," *Applied Intelligence*, vol. 45, no. 3, pp. 808–828, Oct. 2016, ISSN: 15737497. DOI: 10.1007/s10489-016-0786-y. [Online]. Available: https://link-springer-com.tudelft.idm.oclc.org/article/10.1007/s10489-016-0786-y.

[3] S. Mayer, C. Arnet, D. Gankin, and C. Endisch, "Standardized framework for evaluating centralized and decentralized control systems in modular assembly systems," in *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, vol. 2019-Octob, Institute of Electrical and Electronics Engineers Inc., Oct. 2019, pp. 113–119, ISBN: 9781728145693. DOI: 10.1109/SMC.2019.8914314.

[4] J. Liu, P. Mirchandani, and X. Zhou, "Integrated vehicle assignment and routing for system-optimal shared mobility planning with endogenous road congestion," *Transportation Research Part C: Emerging Technologies*, vol. 117, p. 102 675, Aug. 2020, ISSN: 0968090X. DOI: 10.1016/j.trc.2020.102675.

[5] L. Deroussi and S. Norre, "Simultaneous scheduling of machines and vehicles for the flexible job shop problem Solution approach : basic ideas," *International Conference on Metaheuristics and Nature Inspired Computing*, no. February, pp. 2–3, 2010, ISSN: 1432-2218. [Online]. Available: https://www.researchgate.net/publication/265801340.

[6] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Annals of Discrete Mathematics*, vol. 5, no. C, pp. 287–326, Jan. 1979, ISSN: 01675060. DOI: 10.1016/S0167-5060(08)70356-X.

[7] S. M. Homayouni and D. B. Fontes, "Production and transport scheduling in flexible job shop manufacturing systems," *Journal of Global Optimization*, vol. 79, no. 2, pp. 463–502, Feb. 2021, ISSN: 15732916. DOI: 10.1007/s10898-021-00992-6. [Online]. Available: https://link.springer.com/article/10.1007/s10898-021-00992-6.

[8] H. H. Hoos and T. Stützle, *Stochastic local search*, 2007. DOI: 10.1201/9781420010749. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558608726500184.

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959, ISSN: 0029599X. DOI: 10.1007/BF01386390.

[10] M. R. Garey, D. S. Johnson, and R. Sethi, "Complexity of Flowshop and Jobshop Scheduling.," *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, May 1976, ISSN: 0364765X. DOI: 10.1287/moor.1.2.117. [Online]. Available: https://pubsonline.informs.org/doi/abs/10.1287/moor.1.2.117.

[11] A. Rinciog and A. Meyer, "Towards Standardizing Reinforcement Learning Approaches for Stochastic Production Scheduling," *Procedia CIRP*, vol. 107, pp. 1112–1119, 2022. DOI: 10.1016/j.procir.2022.05.117. [Online]. Available: http://arxiv.org/abs/2104.08196.

[12] K. Li, Q. Deng, L. Zhang, Q. Fan, G. Gong, and S. Ding, "An effective MCTS-based algorithm for minimizing makespan in dynamic flexible job shop scheduling problem," *Computers and Industrial Engineering*, vol. 155, p. 107 211, May 2021, ISSN: 03608352. DOI: 10.1016/j.cie.2021.107211.

[13] Q. Zhang, H. Manier, and M. A. Manier, "A modified shifting bottleneck heuristic and disjunctive graph for job shop scheduling problems with transportation constraints," *International Journal of Production Research*, vol. 52, no. 4, pp. 985–1002, 2014, ISSN: 00207543. DOI: 10.1080/00207543.2013.828164. [Online]. Available: http://www.tandfonline.com/loi/tprs20.

[14] M. V. Kumar, R. Janardhana, and C. S. Rao, "Simultaneous scheduling of machines and vehicles in an FMS environment with alternative routing," *International Journal of Advanced Manufacturing Technology*, vol. 53, no. 1-4, pp. 339–351, Mar. 2011, ISSN: 02683768. DOI: 10.1007/s00170-010-2820-2. [Online]. Available: http://link.springer.com/10.1007/s00170-010-2820-2.

[15] Q. Zhang, H. Manier, and M. A. Manier, "A genetic algorithm with tabu search procedure for flexible job shop scheduling with transportation constraints and bounded processing times," *Computers and Operations Research*, vol. 39, no. 7, pp. 1713–1723, Jul. 2012, ISSN: 03050548. DOI: 10.1016/j.cor.2011.10.007. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0305054811002954.

[16] Q. Zhang, H. Manier, and M. A. Manier, "Metaheuristics for Job Shop Scheduling with Transportation," *Metaheuristics for Production Scheduling*, pp. 465–493, Jun. 2013. DOI: 10.1002/9781118731598.ch17.

[17] L. Deroussi, "A Hybrid PSO applied to the flexible job shop with transport," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8472, pp. 115–122, 2014, ISSN: 16113349. DOI: 10.1007/978-3-319-12970-9_13.

[18] H. E. Nouri, O. Belkahla Driss, and K. Ghédira, "Simultaneous scheduling of machines and transport robots in flexible job shop environment using hybrid metaheuristics based on clustered holonic multiagent model," *Computers and Industrial Engineering*, vol. 102, pp. 488–501, Dec. 2016, ISSN: 03608352. DOI: 10.1016/j.cie.2016.02.024.

[19] R. Bellman, "A Markovian Decision Process," *Indiana University Mathematics Journal*, vol. 6, no. 4, pp. 679–684, 1957, ISSN: 0022-2518. DOI: 10.1512/iumj.1957.6.56038. [Online]. Available: https://www.jstor.org/stable/24900506?seq=1#metadata_info_tab_contents.

[20] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, p. 1054, 1998. DOI: 10.1109/TNN.1998.712192.

[21] V. François-Lavet, P. Henderson, R. Islam, *et al.*, "An Introduction to Deep Reinforcement Learning," 2018. DOI: 10.1561/2200000071.

[22] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, May 1992, ISSN: 0885-6125. DOI: 10.1007/bf00992698. [Online]. Available: https://link.springer.com/article/10.1007/BF00992698.

[23] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, May 1992, ISSN: 0885-6125. DOI: 10.1007/bf00992696. [Online]. Available: https://link.springer.com/article/10.1007/BF00992696.

[24] V. Mnih, A. P. Badia, L. Mirza, *et al.*, "Asynchronous methods for deep reinforcement learning," *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, pp. 2850–2869, Feb. 2016. DOI: 10.48550/arXiv.1602.01783. [Online]. Available: https://arxiv.org/abs/1602.01783v2.

[25] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, pp. 1889–1897, Feb. 2015. DOI: 10.48550/arXiv.1502.05477. [Online]. Available: https://arxiv.org/abs/1502.05477v5.

[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," 2017. DOI: 10.48550/arXiv.1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[27] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, Jul. 1991, ISSN: 0163-5719. DOI: 10.1145/122344.122377. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/122344.122377.

[28] J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020, ISSN: 14764687. DOI: 10.1038/s41586-020-03051-4.

[29] S. Racanière, T. Weber, D. P. Reichert, *et al.*, "Imagination-augmented agents for deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 2017-Decem, pp. 5691–5702, 2017, ISSN: 10495258. DOI: 10.48550/arXiv.1707.06203. [Online]. Available: https://arxiv.org/abs/1707.06203.

[30] F. Yi, W. Fu, and H. Liang, "Model-based reinforcement learning: A survey," *Proceedings of the International Conference on Electronic Business (ICEB)*, vol. 2018-Decem, pp. 421–429, 2018, ISSN: 16830040. DOI: 10.48550/arXiv.2006.16712. [Online]. Available: https://arxiv.org/abs/2006.16712.

[31] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018, ISSN: 10959203. DOI: 10.1126/science.aar6404.

[32] T. Anthony, Z. Tian, and D. Barber, "Thinking Fast and Slow with Deep Learning and Tree Search," *Advances in Neural Information Processing Systems*, vol. 2017-Decem, pp. 5361–5371, May 2017, ISSN: 10495258. DOI: 10.48550/arXiv.1705.08439. [Online]. Available: http://arxiv.org/abs/1705.08439.

[33] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature 2016 529:7587*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, ISSN: 1476-4687. DOI: 10.1038/nature16961. [Online]. Available: https://www.nature.com/articles/nature16961.

[34] C. B. Browne, E. Powley, D. Whitehouse, *et al.*, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012, ISSN: 1943068X. DOI: 10.1109/TCIAIG.2012.2186810.

[35] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4212 LNAI, pp. 282–293, 2006. DOI: 10.1007/11871842_29. [Online]. Available: https://link.springer.com/chapter/10.1007/11871842_29.

[36] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Overcoming Exploration in Reinforcement Learning with Demonstrations," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 6292–6299, Sep. 2018, ISSN: 10504729. DOI: 10.1109/ICRA.2018.8463162.

[37] M. E. Aydin and E. Öztemel, "Dynamic job-shop scheduling using reinforcement learning agents," *Robotics and Autonomous Systems*, vol. 33, no. 2, pp. 169–178, Nov. 2000, ISSN: 09218890. DOI: 10.1016/S0921-8890(00)00087-7.

[38] Y. C. Wang and J. M. Usher, "Learning policies for single machine job dispatching," *Robotics and Computer-Integrated Manufacturing*, vol. 20, no. 6 SPEC. ISS. Pp. 553–562, Dec. 2004, ISSN: 07365845. DOI: 10.1016/j.rcim.2004.07.003. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0736584504000572.

[39] X. Chen, X. Hao, H. W. Lin, and T. Murata, "Rule driven multi objective dynamic scheduling by data envelopment analysis and reinforcement learning," *2010 IEEE International Conference on Automation and Logistics, ICAL 2010*, pp. 396–401, 2010. DOI: 10.1109/ICAL.2010.5585316.

[40] C. C. Lin, D. J. Deng, Y. L. Chih, and H. T. Chiu, "Smart Manufacturing Scheduling with Edge Computing Using Multiclass Deep Q Network," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, pp. 4276–4284, Jul. 2019, ISSN: 19410050. DOI: 10.1109/TII.2019.2908210.

[41] S. Luo, "Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning," *Applied Soft Computing Journal*, vol. 91, p. 106 208, Jun. 2020, ISSN: 15684946. DOI: 10.1016/j.asoc.2020.106208.

[42] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and C. Xu, "Learning to dispatch for job shop scheduling via deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 2020-Decem, 2020, ISSN: 10495258. DOI: 10.48550/arXiv.2010.12367. [Online]. Available: https://arxiv.org/abs/2010.12367.

[43] J. H. Wang, P. Cheng Luo, H. Q. Xiong, B. W. Zhang, and J. Y. Peng, "Parallel Machine Workshop Scheduling Using the Integration of Proximal Policy Optimization Training and Monte Carlo Tree Search," *Proceedings - 2020 Chinese Automation Congress, CAC 2020*, pp. 3277–3282, Nov. 2020. DOI: 10.1109/CAC51589.2020.9327564.

[44] P. Tassel, M. Gebser, and K. Schekotihin, "A Reinforcement Learning Environment For Job-Shop Scheduling," Apr. 2021. DOI: 10.48550/arXiv.2104.03760. [Online]. Available: https://arxiv.org/abs/2104.03760v1.

[45] L. Wang, X. Hu, Y. Wang, *et al.*, "Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning," *Computer Networks*, vol. 190, p. 107 969, May 2021, ISSN: 13891286. DOI: 10.1016/j.comnet.2021.107969.

[46] J. Park, J. Chun, S. H. Kim, Y. Kim, and J. Park, "Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning," *International Journal of Production Research*, vol. 59, no. 11, pp. 3360–3377, 2021, ISSN: 1366588X. DOI: 10.1080/00207543.2020.1870013.

[47] B. A. Han and J. J. Yang, "Research on adaptive job shop scheduling problems based on dueling double DQN," *IEEE Access*, vol. 8, pp. 186 474–186 495, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3029868.

[48] A. Rinciog, C. Mieth, P. M. Cheikl, and A. Meyer, *Sheet-Metal Production Scheduling Using AlphaGo Zero*, 2020. [Online]. Available: https://www.repo.uni-hannover.de/handle/123456789/9732.

[49] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, Jan. 2019. DOI: 10.1109/TNNLS.2020.2978386. [Online]. Available: http://arxiv.org/abs/1901.00596%20http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[50] W. Song, X. Chen, Q. Li, and Z. Cao, "Flexible Job Shop Scheduling via Graph Neural Network and Deep Reinforcement Learning," *IEEE Transactions on Industrial Informatics*, Feb. 2022, ISSN: 19410050. DOI: 10.1109/TII.2022.3189725.

[51] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The Handbook of Brain Theory and Neural Networks*, pp. 255–258, 1995. DOI: 10.5555/303568.303704. [Online]. Available: http://www.iro.umontreal.ca/~lisa/pointeurs/handbook-convo.pdf.

[52] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Sep. 2016. DOI: 10.48550/arxiv.1609.02907. [Online]. Available: https://arxiv.org/abs/1609.02907v4.

[53] P. Veličković, A. Casanova, P. Liò, G. Cucurull, A. Romero, and Y. Bengio, "Graph Attention Networks," *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Oct. 2017. DOI: 10.48550/arxiv.1710.10903. [Online]. Available: https://arxiv.org/abs/1710.10903v3.

[54] S. Brody, U. Alon, and E. Yahav, "How Attentive are Graph Attention Networks?," May 2021. DOI: 10.48550/arxiv.2105.14491. [Online]. Available: https://arxiv.org/abs/2105.14491v3.

[55] C. Zhang, Y. Zhou, K. Peng, X. Li, K. Lian, and S. Zhang, "Dynamic flexible job shop scheduling method based on improved gene expression programming," *Measurement and Control (United Kingdom)*, p. 002 029 402 094 635, Aug. 2020, ISSN: 00202940. DOI: 10.1177/0020294020946352. [Online]. Available: http://journals.sagepub.com/doi/10.1177/0020294020946352.

[56] V. M. Dax, J. Li, K. Leahy, and M. Kochenderfer, *Graph Q-Learning for Combinatorial Optimization*, Dec. 2022. [Online]. Available: https://openreview.net/forum?id=srk2FE7q82i.

[57] Q. Wang, Y. Hao, and J. Cao, "Learning to traverse over graphs with a Monte Carlo tree search-based self-play framework," *Engineering Applications of Artificial Intelligence*, vol. 105, p. 104 422, Oct. 2021, ISSN: 0952-1976. DOI: 10.1016/J.ENGAPPAI.2021.104422.

[58] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, "Solving NP-Hard Problems on Graphs with Extended AlphaGo Zero," May 2019. DOI: 10.48550/arXiv.1905.11623. [Online]. Available: http://arxiv.org/abs/1905.11623.

[59] Z. Xing and S. Tu, "A Graph Neural Network Assisted Monte Carlo Tree Search Approach to Traveling Salesman Problem," *IEEE Access*, vol. 8, pp. 108 418–108 428, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3000236.

[60] A. Deichler, *Generalization and locality in the AlphaZero algorithm: A study in single agent, fully observable, deterministic environments*, 2019. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Ab922fec8-8085-4757-a91c-c68d840c03bd.

[61] A. Laterre, Y. Fu, M. K. Jabri, *et al.*, "Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization," 2018. DOI: 10.48550/arXiv.1807.01672. [Online]. Available: http://arxiv.org/abs/1807.01672.

[62] R. Wang, Z. Hua, G. Liu, *et al.*, "A Bi-Level Framework for Learning to Solve Combinatorial Optimization on Graphs," Jun. 2021. DOI: 10.48550/arXiv.2106.04927. [Online]. Available: http://arxiv.org/abs/2106.04927.

[63] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM review*, vol. 59, no. 1, Nov. 2014. DOI: 10.48550/arXiv.1411.1607. [Online]. Available: http://arxiv.org/abs/1411.1607.

[64] M. Innes, E. Saba, K. Fischer, *et al.*, "Fashionable Modelling with Flux," *CoRR*, vol. abs/1811.01457, Oct. 2018. DOI: 10.48550/arXiv.1811.01457. [Online]. Available: http://arxiv.org/abs/1811.01457.

[65] J. Laurent, *AlphaZero.jl: A generic, simple and fast AlphaZero implementation*, 2021. [Online]. Available: https://github.com/jonathan-laurent/AlphaZero.jl.

[66] C. Lucibello, *GraphNeuralNetworks.jl: a geometric deep learning library for the Julia programming language*, 2021. [Online]. Available: https://github.com/CarloLucibello/GraphNeuralNetworks.jl.

[67] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention Is All You Need," Jun. 2017. DOI: 10.48550/arXiv.1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762.

[68] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *32nd International Conference on Machine Learning, ICML 2015*, vol. 1, pp. 448–456, Feb. 2015. [Online]. Available: https://arxiv.org/abs/1502.03167v3.

[69] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-Normalizing Neural Networks," *Advances in Neural Information Processing Systems*, vol. 2017-December, pp. 972–981, Jun. 2017, ISSN: 10495258. [Online]. Available: https://arxiv.org/abs/1706.02515v5.

Glossary

List of Symbols

Abbreviations

$c_{\max}$	Completion time all operations
$\pi_{\boldsymbol{\theta}}(s)$	The policy of the MCTS at state s
$\pi_{mcts}(s)$	The policy of the MCTS at state s
A_{ij}	Set of alternative machines available for o_{ij}
c_i	Completion time of all operations from job i
$c_{o_{ij}}$	Completion time of o_{ij}
J	Set of jobs in the problem instance
K	Set of vehicles in the problem instance
L_V	The loss function for the value function
L_π	The loss function for the policy
M	Set of machines in the problem instance
O	Set of operations in the problem instance
$O_k(o_{ij})$	Vehicle asigned to o_{ij}
$O_m(o_{ij})$	Machine asigned to o_{ij}
o_{ij}	Operation j from a job i
p_{ij}^m	Process time of o_{ij} on m
$R(s)$	Reward for reaching state s
S_p	Ordered set of process operations in the solution
S_t	Ordered set of transport operations in the solution
s_t	The state at time t
$t_{ij}^{mm'}$	Transport time of o_{ij} from m to m'
u_{ij}^{km}	Idle time of o_{ij} using k on m
$V_{\boldsymbol{\theta}}(s)$	The value estimate of s of the neural network
$V_{mcts}(s)$	The value estimate of s of the MCTS
X	Set of node attributes in a graph

X^c	Set of edge attributes in a graph
$x^c_{v,u}$	Edge attributes from node v to u
x_v	Node attributes of node v