BERGISCHE
UNIVERSITÄT
WUPPERTAL

Effective Curriculum Learning Strategies for Deep Reinforcement Learning on the Job Shop Scheduling Problem

Thesis
Zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)
Studiengang Bachelor of Science Informatik
der Bergischen Universität Wuppertal

vorgelegt von
Elias Theis
Matrikelnummer: [redacted]

Betreuer: Constantin Waubert de Puiseau, M.Sc.
Erstgutachter: Prof. Dr.-Ing. Tobias Meisen
Zweitgutachter: Prof. Dr.-Ing. Dietmar Tutsch

Wuppertal, den 16. Februar 2024

Abstract

The job shop scheduling problem is a significant issue in the field of operations research and management science. It pertains to the optimal allocation of capital, goods, and means of production in economic tasks such as education, research, production, warehousing, transport, and sales. However, finding an optimal solution to a large problem of this type can be impractical in the real world.

To find a near-optimal solution much faster, various methods are available ranging from simple priority dispatching rules to genetic algorithms. One modern approach to improving these approximate results is through the use of deep reinforcement learning. An agent learns to build a schedule iteratively. Compared to other approximation methods, this method achieves very good results with a short processing time.

By (dynamically) modulating the difficulty level throughout the learning process, curriculum learning can be used to further improve the results of deep reinforcement learning agents. This technique is already used in many subject areas, but is only very weakly represented in job shop scheduling. This thesis addresses this research gap and develops and analyses different curriculum learning strategies in the context of the job shop scheduling problem.

Considering the disclaimer in section 5.3.1, the relative results show that curriculum learning strategy can be advantageous if it not only starts training on the most difficult instances, but uses them exclusively for training. The difficulty of an instance is determined using a simple priority dispatching rule. The novel approach of expanding the problem space, on the other hand, has a negative effect on the final results.

Contents

Chapter 1

Introduction

In the field of operational research and management science, the Job Shop Scheduling Problem (JSSP) is one of the most explored combinatorial optimisation problems. It is organised as a set of tasks, each consisting of a sequence of operations, to be performed on a variety of machines in the shortest possible time. The formulation of this holds significant importance in optimising the scheduling of real-world tasks. It applies to any economic task that deals with the optimal assignment of capital goods versus means of production, such as education, research, manufacturing, storage, transportation, and sales. [1, 2]

Creating consistently efficient schedules poses a significant challenge. The issue stems from the fact that achieving an optimal schedule often requires expensive and impractical procedures. Since these scheduling problems can be a daily challenge in planning and operation of manufacturing systems, the solution methods have to be able to solve these problems within short amounts of time to be practical in the real-world. Simple heuristics, such as priority dispatching rules (PDR), can be used to obtain near-optimal solutions a lot faster, than optimal solvers. [3, 4]

Optimal solvers already exist [5], but they may not be suitable for larger JSSP instances due to their polynomial runtime. There is a gap between slow and optimal solvers and fast sub-optimal solvers. Therefore, additional research is necessary. A modern approach uses deep reinforcement learning (DRL) to derive improved heuristics automatically from interaction data between an agent and a planning environment. These virtual environments usually generate random problem instances in a random order for the training process. [6, 7]

Curriculum learning (CL) can be a useful method to improve both learning speed

and final performance, as [6, 7] show. This method involves presenting the learning agent with problems of intentionally varying difficulty levels throughout the training process. The difficulty can be varied in many different ways. [6] varies the difficulty by dynamically switching between the different problem sizes in the training process, while [7], in contrast, uses CL to sort the problem instances within a problem size by difficulty. The difficulty of a given instance is determined by the performance of a primitive PDR.

Deep reinforcement learning is a very time-consuming process, which is why optimising the learning speed is of great advantage. In this thesis, different CL approaches are developed, implemented, tested and compared. Specifically, the curricula of [6] and [7] are combined so that the agent switches dynamically between the problem sizes and, in addition, the problem instances within a problem size are ordered according to the solution difficulty. Finally, all approaches are compared and the influence on the overall goals of improving the learning speed and the final performance are compared.

Chapter 2

Fundamentals

This chapter covers the fundamentals necessary for this work. It begins by defining the job shop scheduling problem. Next, it introduces deep reinforcement learning and its relation to the JSSP. Finally, this chapter explains curriculum learning and its relation to Deep Reinforcement Learning and the Job Shop Scheduling Problem. Additionally, two different methods for applying CL to the JSSP are presented.

2.1 The Job Shop Scheduling Problem

The JSSP is a $\mathcal{NP}$-hard combinatorial optimisation problem. $\mathcal{NP}$-hardness (nondeterministic polynomial-time hardness) is a classification of problems that are at least as difficult as the hardest problems in $\mathcal{NP}$. $\mathcal{NP}$ (nondeterministic polynomial time), on the other hand, is the set of decision problems that can be solved in polynomial time by a nondeterministic Turing machine. [8]

The solution methods for the JSSP balance optimality and computation time. Although constraint programming and integer programming can obtain optimal solutions, they are computationally expensive and impractical for larger problem instances. Simple PDRs are much faster and can solve JSSP instances in linear time, while achieving suboptimal solutions.

2.1.1 Problem formulation of the JSSP

The classic deterministic JSSP is defined as a finite set of n jobs $\{J_i\}_{i=1}^{n}$. These have to be processed on a finite set of m heterogeneous machines $\{M_k\}_{k=0}^{m}$. Each job consists of a finite sequence of m operations $O_{i,1} \rightarrow \ldots \rightarrow O_{i,m}$ that have to be

processed in a predetermined order. Each operation $O_{i,j}$ is assigned to a machine M_k for a given processing time $D_{i,j}$. [6]

The total execution time (total time taken to process all jobs) is referred to as the makespan C_{max} and the optimal (minimal) makespan as C^*_{max}. The goal of this problem is to determine the order in which operations are scheduled to minimise the makespan. [1, 9]

The JSSP model works within a set of 3 constraints [6, 3], as outlined below:

($c1$) **No-overlap constraint**: It ensures that each machine shall process only one operation at a time

($c2$) **Non-preemptive constraint**: It states that once the processing of any operation is initiated, it shall not be interrupted before completion

($c3$) **Precedence constraint**: It establishes the order of operations inside a job J_i, where operation $O_{i,j+1}$ shall not be scheduled until the previous operation $O_{i,j}$ of the job J_i is completed

A JSSP instance is referred to with its given size as $n \times m$. For example, an instance with 30 jobs and 15 machines has a problem size of 30×15. To measure the optimality of a solution, the makespan is set in relation to the makespan of an optimal solution. This metric is called the *optimality gap* δ and is defined as follows:

$$\delta = \frac{C_{max} - C^*_{max}}{C^*_{max}} \tag{2.1}$$

An optimal solution has an optimality gap δ of 0%. For example, a solution with a makespan of 14 and an optimal makespan of 10 has an optimality gap of
$\delta = \frac{14-10}{10} = 0,4 = 40\%$. This value is the decisive key figure for measuring how well a method can solve a JSSP.

2.1.2 JSSP Example

The following table is a description of an exemplary 3×3 JSSP instance presented by Yamada et al. [1]. Each of the three jobs comprises one row with the three operations. Each operation is described by a tuple of the assigned machine and the processing time. The precedents of the operations of a job are given by the order of

the operations. For example, the operation $O_{1,2} \rightarrow (2, 3)$ is the second operation of the job J_1 that has to be processed on machine M_2 for 3 time units, after operation $O_{1,1}$ and before operation $O_{1,3}$.

i	$O_{i,1}$	$O_{i,2}$	$O_{i,3}$
1	(1, 3)	(2, 3)	(3, 3)
2	(1, 2)	(3, 3)	(2, 4)
3	(2, 3)	(1, 2)	(3, 1)

Table 2.1: An exemplary 3×3 JSSP instance

The following two figures show two possible solutions for the given scheduling problem as a Gantt chart. The y-axis is divided into three areas for the three machines and the x-axis is divided into discrete time steps. Each operation is shown with the corresponding processing time as the length in the row with the assigned machine. Both solutions meet all constraints, but the second solution (figure 2.2) is superior to the first (figure 2.1) as it utilises the machines more efficiently and results in a shorter makespan.

The first solution is created with an algorithm that iteratively selects a random operation in $n \cdot m$ steps and inserts it into the production plan. It was constructed in less than 0.6ms. The second solution is an optimal solution created with the *SP-SAT solver by OR-Tools* [5] in less than 7ms.

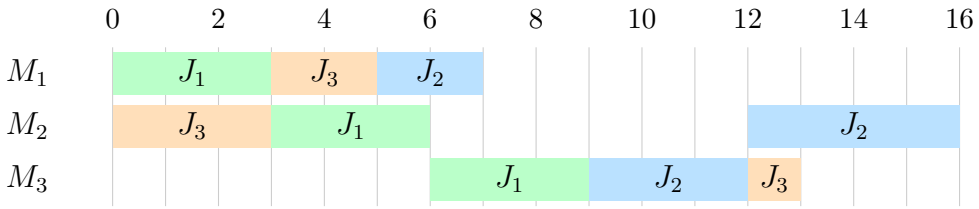

Figure 2.1: A randomly generated solution of the JSSP instance in table 2.1.2

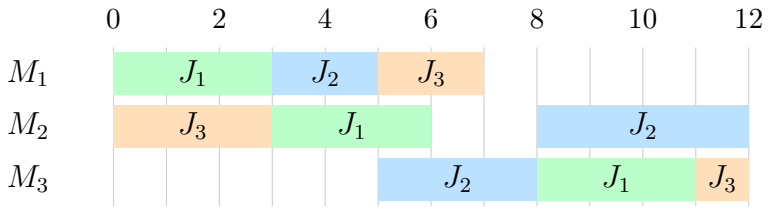

Figure 2.2: An optimal solution of the JSSP instance in table 2.1.2

The second (optimal) solution results in a makespan of $C_{max} = C^*_{max} = 12$, while the first (randomly generated) solutions results in a makespan of $C_{max} = 16$. Therefore, the first solution has an optimality gap of $\delta = \frac{16-12}{12} \approx 33.33\%$.

2.1.3 Datasets

There are different sets of JSSP instances with their corresponding best known solutions. The sets from Taillard (TA) [10] and Demirkol [11] are often used to evaluate different methods of solving the JSSP. Taillard's set contains 80 instances with problem sizes ranging from 15×15 up to 100×20. The machine orders and processing times are sampled from a normal distribution from the interval $1 - 100$. Demirkol's set contains 80 instances with problem sizes ranging from 20×15 up to 50×20.

Proposals exist for more advanced datasets that take diversity in jobs and operations into account, as discussed by Kemmerling et al. [12]. However, older methods logically do not incorporate these datasets into their evaluation processes due to their up-to-date nature. In this work, only the datasets from Taillard [10] and Demirkol [11] are used because reevaluating existing methods with this new dataset can be very time-consuming.

2.1.4 Solution Methods

In general, algorithms created to solve scheduling problems can be classified into two main categories: exact algorithms and approximate algorithms. A number of approximation algorithms have been developed in addition to exhaustive search algorithms based on branch and bound methods. The most commonly used ones in practice are based on PDRs and active schedule generation [13]. *Shifting bottleneck* (SB) [14], a more sophisticated method, has been shown to be very successful

Exact Algorithms	**Approximate Algorithms**
- algorithms based on branch and bound methods	- Priority Dispatching Rules (PDRs) [13]
- SP-SAT solver by OR-Tools [5]	- Shifting Bottleneck [14]
	- Active Schedule Generation
	- Genetic Algorithms [15]
	- Machine Learning (section 2.2)

Table 2.2: Incomplete list of solution algorithms and their classification

[14]. Stochastic approaches, such as *simulated annealing* (SA), *tabu search* [16], and genetic algorithms (GAs), are also used. [1, 2]

2.2 Deep Reinforcement Learning

In order to define deep reinforcement learning, more basic terms must first be introduced, starting with machine learning - a subcategory of the very general field of artificial intelligence (AI). Machine learning (ML) involves the study and development of algorithms that enhance their performance through experience, focusing on inducing models with minimal human intervention, primarily relying on data. When addressing a problem, individuals typically create a model or program capable of solving it. However, in many cases, accurately formalising the problem can be challenging. For example, considering the task of creating a program to distinguish between images of cats and dogs. Traditional approaches might involve manually identifying distinguishing features of each animal, a process that requires expertise and time. In contrast, a machine learning algorithm simplifies the task by allowing the system to automatically generate a model from the available data. This approach presents numerous opportunities for practical applications, especially in scenarios where formalising or programming a solution is challenging, yet relevant data is accessible. [17]

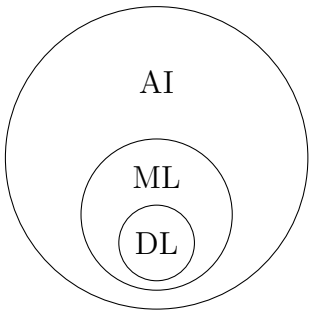

Figure 2.3: Venn diagram representing the relationships between artificial intelligence (AI), machine learning (ML) and deep learning (DL)

There a three basic ML paradigms - supervised learning (SL), unsupervised learning (UL) and reinforcement learning (RL). SL tries to generalise the mapping from input values to the given corresponding output values. These could be RGB values from an image, for example, while the output is an one-hot-encoded array of classes. Both, the input and output values can be discrete or continuous. UL focuses on uncovering patterns or structures within data without labelled outputs. In unsupervised learning, the algorithm explores the inherent relationships and associations within the input data to reveal hidden insights. This paradigm is particularly useful when the task involves discovering underlying structures, clustering similar data

points, or reducing the dimensionality of the input space. [18]

On the other hand, RL revolves around the concept of learning through interaction with an environment. In RL, an agent takes actions in an environment, receives feedback in the form of rewards or penalties, and adjusts its strategy to maximise cumulative rewards over time. The strategy arises from the implementation of the rules from the policy. RL algorithms employ the approach of acquiring knowledge through sequences of actions, observations, and rewards in the environment. RL has demonstrated remarkable success in diverse tasks, ranging from robotics [19] to resource allocation [20]. [21, 17]

The integration of deep learning (DL) techniques into the framework of reinforcement learning has given rise to the paradigm of deep reinforcement learning (DRL). Deep learning involves the use of neural networks to automatically learn hierarchical representations of data, enabling more effective feature extraction and abstraction. In the context of reinforcement learning, the incorporation of deep learning allows the agent to handle high-dimensional input spaces, such as raw pixel data from images or raw sensor inputs. This capability significantly broadens the scope of problems that can be addressed, as DRL excels in tasks where traditional RL methods struggle due to the complexity and richness of the input information.

2.3 Curriculum Learning for Deep Reinforcement Learning

Neural networks form the fundamental building block for most deep learning applications and are modelled on the human brain. It therefore makes sense to try to replicate not only the structure of the brain, but also to model the way it learns. Humans generally learn with increasing complexity over time. They start with simple problems and concepts and then move on to difficult and complex ones. This intuitive approach can be found in the curricula of school systems all over the world. [22]

CL can lead to two fundamental advantages: increasing the convergence speed of the training process and better final performance. The empirical results from Soviany et al. [22] show the clear benefit of using CL instead of a random mini-batch approach.

The genesis of CL can be traced back to the work of Bengio et al. [23], who introduced the concept in 2009, formalizing the method within the domain of machine learning. Subsequent to this seminal work, various categories of CL have been formulated. Soviany et al. [22] have systematized these diverse methods: Vanilla CL, Self-paced learning (SPL), Balanced CL (BCL), Self-paced CL (SPCL), Teacher-student CL, Implicit CL (ICL), Progressive CL (PCL).
These classes are largely independent of each other and classify different aspects. For example, if a CL approach is not explicitly defined but arises implicitly from other rules, it would fall under the ICL class. However, this does not exclude the possibility that other classes may also apply, as they are independent. If an approach uses a static pace, it falls under the Vanilla-CL class. On the other hand, if a dynamic pace is used, the approach can be classified as a SPL. The use of a dynamic model results in classification as a PCL. If there are additional constraints, it is considered a BCL. If two models form a teacher-student architecture, it is a teacher-student CL. Soviany et al. [22] even define an additional class for the combination of static and dynamic speed, as this occurs frequently: SPCL. [22]

These classes can be viewed as independent dimensions rather than disjoint categories. Clear classification may be challenging under certain circumstances, and the classes serve more as a framework. This description of the classes is incomplete, and other factors may play a role that are harder to define. In the context of this thesis, a subset of these methods is explored in greater detail. [22]

Vanilla CL or just CL is the name of the method Bengio et al. [23] coined in 2009. They first introduced the approach of increasing the difficulty during training [22]. In many cases, CL is only mentioned if the speed at which the difficulty is adjusted is already prescribed. If the curriculum adjusts the pace of increasing difficulty based on the model's performance, this is known as self-paced learning (SPL) or self-paced curriculum learning (SPCL). [24]

Self-paced curriculum learning (SPCL), a paradigm combining SPL and CL, involves the joint utilisation of predefined criteria and learning-based metrics to establish the training order of samples. Jiang et al. [24] initially introduced this paradigm, employing it in the domains of matrix factorisation and multimedia event detection. [22]

Furthermore, even more methods and combinations can be created. In addition

to the methods presented so far, a distinction can also be made between two basic frameworks: data-level curriculum learning and model-level curriculum learning. A distinction is made here between changing the training data and changing the model itself during training [22]. A combination of the two is also possible. However, this work only deals with data-level CL.

The difficulty with any CL approach is to develop a curriculum that supports the model in learning - increased speed of learning and improved final results are the main objectives.

2.4 Generalisation with RASCL

This section summarises the approach of Iklassov et al. [6] from the paper *On the Study of Curriculum Learning for Inferring Dispatching Policies on the Job Shop Scheduling*. It is a relatively recent paper from November 2022, which focuses on the generalisation of dispatching rules on the JSSP. Generalisation, in this context, refers to the ability to solve different sizes of the JSSP. This work forms the basis of this thesis. The proposed model architecture is adopted unchanged, but the training procedure, including the curriculum, is fundamentally changed and discussed later in chapter 4. The implementation available online[1] forms the code basis for this thesis.

An established method for enhancing generalisation is to employ CL, which involves training on progressively challenging instances. Nevertheless, studies from Lisicki et al. [25] suggest that this approach may encounter the challenge of catastrophic forgetting when applying learned skills to various problem sizes. In order to tackle this issue, Iklassov et al. [6] proposed an innovative strategy called *Reinforced Adaptive Staircase Curriculum Learning* (RASCL). This approach dynamically modulates the difficulty level throughout the learning process, allowing for a revisit of the most challenging instances to prevent performance deterioration.

[1] `https://github.com/Optimization-and-Machine-Learning-Lab/Job-Shop/tree/main_nips`

2.4.1 Method

Iklassov et al. [6] address the JSSP as a Markov Decision Process (MDP), wherein the model iteratively constructs a solution based on the operations that are yet to be scheduled, as well as on the state of the problem resolution. The model is equivariant with respect to job information and size-agnostic, enabling Iklassov et al. [6] to train the model on different problem sizes. This attribute is key for the generalisation study conducted in their paper.

To complete a schedule of a JSSP instance, $n \cdot m$ dispatches have to be assigned sequentially. The autoregressive model solves a problem x, by iteratively dispatching one operation at a time according to its scheduling policy. At a decision step t, the model takes the instance definition and the state s_t as inputs for the resolution process, generating a probability distribution for the action at at that time. The chosen operation is then scheduled, resulting in the acquisition of the next state s_{t+1}. This process iterates until all pending operations are scheduled. The solution y is defined as the sequence of selected actions $a_1, a_2, ..., a_{nm}$. Where $a_t \in \{1, \ldots, n\}$ is the index of scheduled job J_i. This means that each of the n indices should occur exactly m times in a_t.

The reward of the DRL agent is defined as the difference of the makespan before and after the dispatch of an operation. The cumulative reward is therefore equal to the negative makespan of the plan. [6]

2.4.2 Model Architecture

The proposed architecture involves a two-stage process, starting with deep learning input preprocessing and followed by policy inference using actor-critic networks. The model's input is divided into two branches, namely static and dynamic components. The static input encompasses information regarding all operational blocks within a given job J_i and is incorporated into the upcoming operation through a reversed Long Short-Term Memory network (rLSTM) [26]. This rLSTM propagates information from the final operation (indexed as m) of job J_i backward to the current operation (indexed as j). [6]

Simultaneously, the dynamic input captures details about the current environment state, including operations currently in progress, machine statuses, and re-

maining processing times. The embeddings from both static and dynamic branches are concatenated into multidimensional representations, which are then input into actor-critic networks for reward calculation of the current state and to suggest the next dispatch. [6]

To ensure size-agnostic functionality, the model operates on multidimensional embeddings of actual 3-dimensional operations. Specifically, for scheduling an operation at any given time, the architecture employs a reverse LSTM that propagates information from the last operation in a job to the current operation. This reverse LSTM output is then fed into a *set2set* module [27], which proves particularly useful in aggregating information from all embeddings generated by the rLSTM. Importantly, this module disregards the positioning indices of operations within their jobs, facilitating the circulation of updates between jobs. [6]

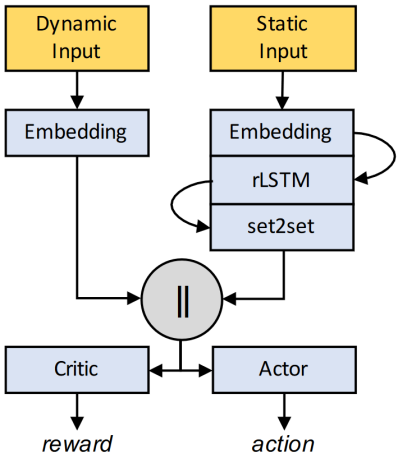

Figure 2.4: Proposed RL architecture from [6]

2.4.3 Inference Strategies

When dealing with choosing an action based on a probability distribution, as mentioned in subsection 2.4.1, there is a need for a strategy. The simplest strategy, called *greedy*, always chooses the action with the highest probability. However, as [28] and [29] show, it can be very useful to choose more complex strategies to improve the results. [6]

The second strategy presented by [6] is *sampling*. During sampling, strategy actions are selected randomly from the policy distribution. Consequently, the suggested actions for the same state and instance may vary. To ensure the superiority of

the sampling strategy over greedy, it necessitates fine-tuning over a certain number of training iterations. [6]

POMO is another strategy. *Policy Optimisation with Multiple Optima* (POMO) integrates elements from both greedy and sampling strategies. Starting at the initial state s_0, it generates multiple actions $\{a_0^{(1)}, a_0^{(2)}, \ldots\}$ and subsequently creates various possible trajectories $\{s_1^{(1)}, s_1^{(2)}, \ldots\}$, evolving these trajectories in a greedy manner. In a study by the authors [30], it is shown that, for certain combinatorial problems, this approach yields superior results compared to the sampling strategy. [6]

Finally, the *Beam Search* strategy is presented. This strategy [31, 32] follows a greedy approach by selecting k actions $\{a_0^{(1)}, \ldots, a_0^{(k)}\}$ at the initial state s_0, thereby creating k trajectories $\{s_1^{(1)}, ..., s_1^{(k)}\}$. Subsequently, for each trajectory, a new set of k actions is unfolded, expanding the total number of available actions to $k \times k$. To prevent an excessive increase in the number of explored trajectories, the likelihood of each action is computed. The strategy then greedily opts for the next k actions with the highest likelihood. In this manner, *Beam Search* systematically explores the k most likely trajectories at each step. Greedy is just a special case of *Beam Search* if k is set to 1. [6]

Tests from Iklassov et al. [6] showed that, when employing three possible trajectories for both POMO and *Beam Search*, both strategies exhibit approximately similar results. However, the greedy algorithm performs slightly worse as it represents a special case of *Beam search*. Notably, the sampling strategy outperforms all others across all test sizes, and is used for training in all experiments. [6]

2.4.4 Curriculum Learning Strategies

To develop curriculum learning strategies for the JSSP, the basic assumption is made that a larger problem size is more complex and requires more time to train a model. The authors present four different CL approaches, including a novel approach that yields significantly better results. These approaches are briefly explained below.

Incremental Curriculum Learning (ICL) involves sequentially training the model on progressively challenging levels. Specifically, in the context of the JSSP, a model is trained on each problem size over a fixed number of learning steps. However, the inherent limitation of this straightforward strategy is the model's susceptibility to catastrophic forgetting when exposed to instances of consecutive complexity grades

[25]. Additionally, this approach results in a growing number of models, eventually equating to the number of relevant problem sizes, rather than having a single universal policy. [6]

Uniform Curriculum Learning (UCL) involves selecting a problem of random size from uniformly distributed difficulty levels at each iteration. This approach ensures that the model is randomly exposed to various training levels, promoting a policy that is inherent to different problem sizes. Nevertheless, according to the findings in [25], it appears that the learning process is more effective when focused on smaller-size tasks rather than uniformly sampling from tasks of all sizes. This suggests that a uniform distribution of difficulty levels may not be the most advantageous learning strategy. [6]

Adaptive Staircase Curriculum Learning (ASCL) introduces a controlled level of flexibility in determining the next JSSP problem size for training. It was introduced to RL in [25]. The problem sizes are assigned a sequential order, so that these variables can be seen as a staircase ascending in difficulty, ranging from l_{min} to l_{max}. The current level l is set to l_{min} in the beginning. At each time step t there are 3 possible options the strategy can choose from:

1. Advance to the next problem size $l \leftarrow l + 1, \quad \text{if } l + 1 \leq l_{max}$
2. Stay at the current problem size l
3. Regress to the previous problem size $l \leftarrow l - 1, \quad \text{if } l - 1 \geq l_{min}$

The strategy, however, solely evaluates the model's behaviour for adjacent difficulty levels, specifically those with a step difference of 1. This method does not assess the model's generalisation performance across other difficulty levels. It can therefore easily suffer from catastrophic forgetting. [6]

One of the main contributions from [6] is RASCL. A strategy that tries combines the advantages of different strategies: the learning speed of ASCL and the stability of UCL. In each i^{th} iteration, the RASCL algorithm compares future rewards to those of the optimal solution. To be precise, it normalises optimality gaps for each difficulty level $l' \leq l$ and transforms them into probabilities, denoted as g. This process assigns higher probabilities to larger gaps, with each probability corresponding to a specific problem size. The agent undergoes u iterations of learning at level l, after which the gap to optimal is compared to a threshold value t_{opt}. If the gap closes, RASCL adopts the next problem size for training like ASCL ($l \leftarrow l + 1$). However,

if the gap does not close, every b steps, the algorithm samples a smaller level l' from the distribution of gaps. This algorithm is described in more detail as pseudo code in section 4.2.1.

Revisiting and anchoring the most challenging difficulty levels, the proposed algorithm reinforces a recently learned policy. Unlike ASCL, it enables backtracking to all problem sizes previously encountered by the agent. Derived from the former, RASCL directs the model's attention to the sizes of JSSP where its performance is poorest while maintaining flexibility to revisit any size. [6]

2.4.5 Results

Iklassov et al. [6] presented methods that reduced the average optimality gaps from 19.35% to 10.46% on the Taillard instances [33] relative to state-of-the-art works on JSSP [32, 34]. Similarly, for the Demirkol instances, a reduction from 38.43% to 18.85% was achieved. [6]

2.5 CL based on instance difficulty

This section addresses the work of Waubert de Puiseau et al. [7]. The authors claim that in real-world scenarios, the number of machines is often constant and existing work focuses only on the variability of problem sizes. Therefore a different approach to CL is used in order to optimise the training speed as well as the final results. They investigate the effect of more granular CL within a constant problem size, as these have not yet been investigated. [7]

The primary distinction from [6] is the fundamental CL approach to the JSSP. In [7], the JSSP instances are ordered within a problem size, while [6] focuses on the order of the problem sizes themselves. As a result, a distinct model is trained for each necessary problem size. CL is utilised by dynamically incorporating variations in difficulty within a constant problem size into the learning process. [7]

2.5.1 Method

Waubert de Puiseau et al. [7] extend the method and framework from Zhang et al. [34]. Their method specifically adapts the interaction logic of the DRL agent with

the simulation, the action evaluation signal, the input formulation, and the network architecture. As in [6], the difference between the makespan before and after the dispatch is used as the reward. [7]

There is no explicit mention of inference strategies. However, it is very likely that *greedy* is used, since the agent is "choosing from the list of still unfinished jobs in each iteration step" [7]. A more complex strategy such as POMO or Beam Search would most likely be mentioned. [7]

2.5.2 Curriculum Learning Strategy

Creating a CL strategy similar to RASCL may seem straightforward, as the different JSSP problem sizes implicitly provide a ranking of difficulties. However, to determine the difficulty of instances within a problem size, a custom feature is essential. Given that instances of the same problem size inherently share the same computational complexity, the authors rely on a characteristic determined by their proficiency in solving instances through a well-established set of rules, namely PDRs. Waubert de Puiseau et al. [7] create the feature *difficulty to solve* (DTS) which is characterised by the makespan achieved by the most competitive PDR on a given problem instance. JSSP instances with a shorter than average makespan are defined as *easy* tasks and instances with a longer than average makespan are considered as *hard* tasks. When solving 40,000 randomly generated 6x6 JSSP instances with six frequently used PDRs, *most tasks remaining* (MTR) achieves the lowest makespans on average with an optimality gap of about 16%. The machine orders and processing times are sampled from a normal distribution from the interval $0 - 100$, as in Zhang et al. [34]. [7]

After generating the task, the authors then focus on sequencing, determining the order in which to present the training experience. To cover more than just the intuitive approach of ever-increasing complexity, the authors split the generated instances into easy and hard halves. By adding the option to reverse the order of any half, four possible halves are created, referred to as curriculum elements (CEs). Combining two CEs results in 16 possible curricula that can be considered. [7]

The experiments are structured to attribute differences in agent behaviour and performance solely to the training curricula. Separate RL-agents are trained for each of the 16 curricula until all instances are presented once. As a baseline, three RL-

agents are trained on randomly shuffled data with varied initial seeds. Testing occurs on a fixed dataset of 1000 problem instances after every 2000 training instances. To enhance statistical significance, three different training datasets with varying seeds are sampled. These experiments are conducted independently on three datasets. [7]

2.5.3 Results

The results show a swift decrease in the mean optimality gap of the three validation instances of all agents. This dip occurs in the training process from 2,000 to 6,000 instances, followed by gradual convergence to higher optimality gaps. Notably, over 70% of agents find their global minimum in the initial dip, showcasing the early formulation of a successful strategy. However, they do not revert to it, instead converging towards a higher optimality gap. The best results are achieved by the agents that were have trained on the hardest data first (i.e. the harder half sorted by DTS in descending order). The worst results are achieved by the agents who first trained on the simple instances in ascending order. The second CE of each curriculum has no significant impact on the performance of the agents. However, a small positive influence of hard instances at the beginning of the CE can still be observed. Overall, Waubert de Puiseau et al. [7] achieve 3% (0.5%p) better optimality gaps compared to training with random instances.

Chapter 3

Related Work

This chapter examines previous research on the job shop scheduling problem, deep reinforcement learning, and curriculum learning. The objective of this thesis is to identify effective curriculum learning strategies for deep reinforcement learning in the context of the job shop scheduling problem, which involves the integration of these three areas. However, the following literature review reveals that there has been limited research in this field so far. Only two papers, [6] and [7], have been published on this topic.

3.1 Literature Review

There are many works on all three of these topics. But since a search for a combination of the three topics mentioned only yields the two works already mentioned, the search is more generalised to the combination of reinforcement learning and the JSSP. The main source of literature for this work is Google Scholar (GS) [35]. This literature review considers the sorting of the results in three different ways:

1. Google Scholar's [35] *relevance*
2. Number of citations
3. Number of citations per year since publication (average of all years)

It is important to note that GS shows the number of citations, but does not offer to sort by them. However, open-source software from Wittmann [36] makes it possible to use GS in such a way that sorting by citations or citations per year is also possible. To show that CL has only very rarely been applied to DRL for the JSSP, the table 3.1 shows the 5 most frequently cited works when searching GS for "*job shop*

Citations	Title	Year
623	A reinforcement learning approach to job-shop scheduling [37]	1995
354	Dynamic job-shop scheduling using reinforcement learning agents [38]	2000
325	Optimisation of global production scheduling with deep reinforcement learning [39]	2018
268	Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning [40]	2020
247	A reinforcement learning approach to parameter estimation in dynamic job shop scheduling [41]	2017

Table 3.1: 5 most cited papers on Google Scholar [35] for "*job shop scheduling reinforcement learning*".

scheduling reinforcement learning". Regardless of whether one selects the papers by GS's *relevance*, number of citations, or number of citations per year, none of the resulting papers use curriculum learning. The two papers covering all topics were published in the last few years (2022 and 2023), suggesting ongoing research in this area.

3.2 Differentiation from the closest work

The work of Iklassov et al. [6] and Waubert de Puiseau et al. [7] deal with a very similar topic to this work. Nevertheless, there are major differences.

The authors of [6] base the development of the curriculum on the assumption that smaller problem sizes of the JSSP are easier to solve than larger ones. The presented approach RASCL represents (also independent of the JSSP) a method with which DRL agents can learn more effectively in order to achieve better final results. However, the solvability of all instances within a problem size is regarded as equivalent. The authors of [7], on the other hand, focus on finding differences in the difficulty of solving JSSP instances within a problem size. This property is manifested in a feature and serves as a basis for the development and investigation of different curricula. However, the DRL agents are only ever trained on one JSSP problem size. The training of an agent on several problem sizes is not carried out. This work builds on the implementation of [6] and the findings of [6] and [7] to develop even more sophisticated curricula.

Chapter 4

Methods

This chapter discusses the used methods, including the baselines, termination criteria, and CL strategies developed in this thesis. The implementation of this work is based on that of [6], and the methods explained in section 2.4.1 serve as a basis. Further details on the experimental setup are provided in section 5.1.

To provide a better overview of the various problem sizes, a few sets of problem sizes are defined in advance. The 10 standard problem sizes (SPS) ranges from 6×6 to 100×20. These variables can be subdivided into smaller groups, as shown in Table 4.1. The set of 8 problem sizes of TA is a subset of the SPS. TA is only missing the two smallest sizes 6×6 and 10×10.

Small problem sizes	$6 \times 6,\ 10 \times 10,\ 15 \times 15,\ 20 \times 15$
Medium problem sizes	$20 \times 20,\ 30 \times 15,\ 30 \times 20,\ 50 \times 15$
Large problem sizes	$50 \times 20,\ 100 \times 20$

Table 4.1: 10 standard problem sizes (SPS) divided into 3 size classes

4.1 Baseline strategies

Two CL strategies are used as a basis for comparison. On the one hand, the RASCL algorithm is used unchanged as a comparison; on the other hand, an agent without a curriculum is also trained (NOCL). These two strategies are shown as pseudocode. The RASCL algorithm from [6] is rewritten in adjusted notation to be consistent with the other strategies and aid comparability.

4.2 Termination Criteria

All CL strategies use the same termination criteria as RASCL. The algorithm terminates, when it increases the level l over l_{max}. There are two possible criteria to increase the level. If any of these are met, the level l will be increased:

($i1$) the level l is unchanged for 300 epochs

($i2$) the threshold value for the percentage gap with the optimal solution of the *testdata* is exceeded

The first incrementing criterion ($i1$) is trivial and prohibits the indefinite stagnation at a single problem size. This results in each agent being trained for a maximum of 3000 (10 SPS $\cdot$ 300) epochs. The second criterion is more complex and defined in greater detail by the following inequality:

$$C_{max}[l] \leq \overbrace{C^*_{max}[l] \cdot (1 + t_{opt})}^{C^{\Delta}_{max}[l]} \tag{4.1}$$

with

$C_{max}[l]$	makespans of level l of *testdata* (D1, section 5.1)
$C^*_{max}[l]$	optimal makespans of level l of *testdata* (D1)
t_{opt}	optimality threshold (15%)

This termination criterion is used for all new CL approaches presented here, as it is dynamic and yet has no direct influence on the learning behaviour of the agents. At this point, it should be noted that [6] does not mention the first criterion. However, since it is utilised in the published code, it is also adopted in this paper.

4.2.1 Presentation of the strategies

Each strategy developed in this thesis is presented in different ways. First, a strategy is described and then precisely defined by a pseudocode. In addition, the presentation of some strategies is supported by further illustrations. For example, a figure is used which shows the distribution of the DTS of the instances used as a boxplot for the first 10 epochs of training. This is intended to provide a simple overview of how the strategies generate the training data.

The inputs of the pseudocodes are explained below. The used values of the parameters can be found in the appendix. The parameters l_{min} and l_{max} are positive

integers that denote the lower and upper limits for the RASCL levels respectively. The parameter u is a rate for updating the percentage gaps g. The data loader *testdata* represents the data set (D1) mentioned in the section 5.1. The input θ represents all model parameters in a simplified form. The constant t_{opt} is a threshold value for ascending a RASCL level. The list of *batchsizes* may seem unexpected in this context. It describes not only the batch sizes for training the actor and critic models but also the number of instances generated for each training step. It should be noted that the pseudocode uses the python syntax [42, 43] for slicing arrays.

Since all strategies have the same termination criterion, a complete pseudocode for each strategy would generate many repetitions. To avoid these repetitions, the method $PDTC(i, l, \theta) \rightarrow g, l$ is defined first, which is used by the other strategies. This partition improves the readability of the algorithms.

Algorithm 1 $PDTC(i, l, \theta) \rightarrow g, l$
Performance Distribution and Termination Check

Inputs: positive integers l_{max}, u, dataloader *testdata*,
non-negative float optimality threshold t_{opt},
opt. makespans of *testdata* C^*_{max}

if $i\%u == 1$ **then** ▷ every *uth* epoch
  Update C_{max} array of makespans on *testdate* of θ
  for $k = l_{min}, \ldots, l_{max}$ **do**
    $C^{\Delta}_{max}[k] \leftarrow C^*_{max}[k] \cdot (1 + t_{opt})$ ▷ add 15% to opt. makespan
    $g[k] \leftarrow (C_{max}[k] - C^{\Delta}_{max}[k]) \setminus C^{\Delta}_{max}[k]$ ▷ calculate gap to $C^{\Delta}_{max}[k]$
  end for
  if $C_{max}[l] \leq C^{\Delta}_{max}[l]$ **or** l unchanged for 300 epochs **then**
    $l \leftarrow l + 1$ ▷ increase level
    if $l > l_{max}$ **then**
      Terminate *training*
    end if
  end if
  $g \leftarrow g[l_{min} : l]$ ▷ limit distribution based on l
  $g \leftarrow \text{softmax}(g)$ ▷ apply softmax to g
end if
return g, l ▷ return performance distribution and level

Algorithm 2 Reinforced Adaptive Curriculum Learning (RASCL) algorithm

Inputs: initialised network parameters θ, array of *batchsizes*, positive integer l_{min}

$l \leftarrow l_{min}$
for $i = 1, \ldots, 3000$ **do**
    $g, l \leftarrow PDTC(i, l, \theta)$ ▷ see algorithm 1
    Sample current size s following distribution of g
    Generate ($batchsizes[s]$) *instances* of size s
    Update θ on *instances*
end for
return *trained parameters* θ

Algorithm 3 No Curriculum Learning (NOCL) algorithm

Inputs: initialised network parameters θ, array of *batchsizes*, positive integer l_{min}

$l \leftarrow l_{min}$
for $i = 1, \ldots, 3000$ **do**
    $g, l \leftarrow PDTC(i, l, \theta)$ ▷ see algorithm 1
    ▷ note that g is not used here
    Sample current size s following equal distribution over *SPS*
    Generate ($batchsizes[s]$) *instances* of size s
    Update θ on *instances*
end for
return *trained parameters* θ

4.3 Combination of RASCL and DTS

This strategy class adds a second dimension to the curriculum by combining the successful approaches of [6] and [7]. RASCL lays the foundation for the choice of problem size, while a new feature is introduced to sort instances within a problem size: DTS (as in section 2.5.2).

There is a fundamental hurdle to overcome. Since the models make no difference when sorting instances within a training batch, a different solution is required to sort instances within a problem size. A new parameter is introduced for this purpose. The new parameter *stepsize* determines the number of consecutive epochs on which the model is trained at one size. By default, RASCL selects a new size from the distribution for each training step. However, the strategies in this class remain at the same size for *stepsize* epochs. This repetition allows for the incorporation of the DTS-based application. At the start of each repetition, a set of *stepsize*-fold instances is randomly generated, solved with a PDR, and then sorted. These instances are then divided into *stepsize* subsets and used for training in that order.

The authors of [7] chose MTR as the PDR for measuring the difficulty of an

instance, since MTR has the lowest average optimality gap compared to other frequently used PDRs. In the following, a differentiated check is also made as to whether MTR also performs best in each of the 10 SPS and not just on average. As shown in the appendix, MTR has the smallest makespans in each of the 10 SPS compared to the other PDRs considered. The table 4.2 shows the average Makespans of 5 PDRs on average across all problem sizes. Thus, MTR is also used in this work to determine the DTS.

PDR	Random	SPT	LPT	MTR	LRPT
Average Makespan	39.4	168.0	168.2	**27.6**	31.8

Table 4.2: Average makespan per PDR on 1,000 randomly generated instances per SPS

The following two sections present two CL strategies of this class. The first strategy (CCL) implements the above-mentioned approach in a primitive manner, while the second strategy (SSCL) builds on the first and pursues a more sophisticated goal.

4.3.1 Combined Curriculum Learning (CCL)

This strategy follows the approach already mentioned and is called *Combined Curriculum Learning* (CCL). According to the findings of [7], it can be beneficial to train on the hard instances first. For this reason, the inverted strategy is also introduced: *Inverted Combined Curriculum Learning* (ICCL). Only the order is inverted when sorting according to the DTS, while the rest remains identical.

Figure 4.1 illustrates how the strategy works by showing the DTS of the generated instances as a box plot. Each box comprises 128 instances. With a *stepsize* of 4, the periodically repeated increase in the DTS can be recognised. As the *PDTC()* method has not yet increased the level in the first 10 epochs ($l = l_{min}$), all the instances shown are of size 6×6 (first SPS). This results in a DTS range of approximately 4 to 10. Furthermore, it is easy to see that the values are strongly centered in the middle. Very high and very low values are only rarely represented, suggesting that the distribution is similar to the normal distribution.
An equivalent figure for the inverted variant of the strategy is not shown here, as only the order of the 4 boxes within a repetition would be inverted.

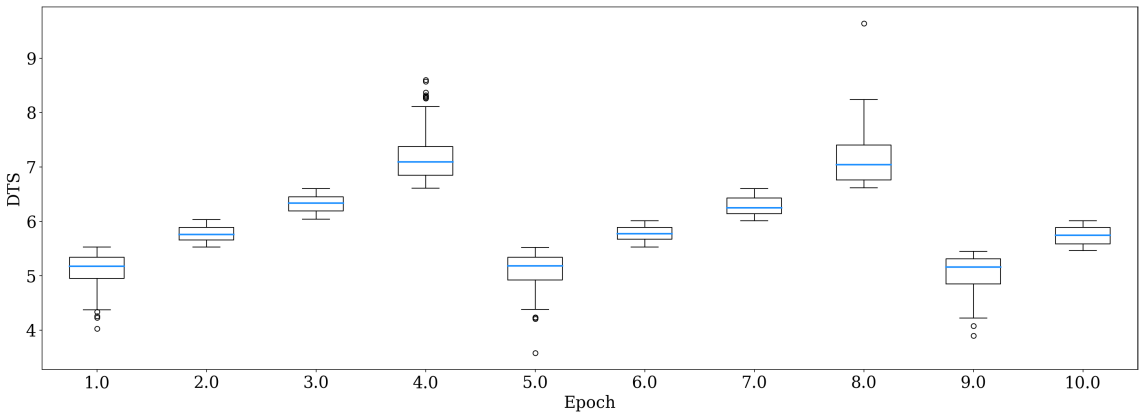

Figure 4.1: Visualisation of CLL with 4 levels and 128 instances per batch

The following pseudocode describes the strategy. The ICCL strategy is equivalent, with the only difference that the sort order in row 7 is inverted.

Algorithm 4 Combined Curriculum Learning (CCL) algorithm

Inputs: initialised network parameters θ, array of *batchsizes*, positive integer l_{min}, *stepsize*

1: $l \leftarrow l_{min}$
2: **for** $i = 1, \ldots, 3000$ **do**
3: $g, l \leftarrow PDTC(i, l, \theta)$ ▷ see algorithm 1
4: **if** $i\%stepsize == 0$ **then**
5: **Sample current size** s following distribution of g
6: **Generate** $(batchsizes[s] \cdot stepsize)$ *instances* of size s
7: **Sort** *instances* by *MTR* makespans ascending
8: **Split** *instances* into *stepsize* blocks $\rightarrow b$
9: **end if**
10: **Update** θ on $b[i\%stepsize]$
11: **end for**
12: **return** *trained parameters* θ

4.3.2 Spaced Sampling Curriculum Learning (SSCL)

The *Spaced Sampling Curriculum Learning* strategy is based on CCL and focuses on the selection of instances and not just their order during training. If one solves a large number of instances with MTR and examines the distribution of the makespans (figure 4.2), it clearly approaches a slightly skewed normal distribution. This means that there are many instances that are moderately difficult to solve according to DTS, but very few instances that are either very easy or very hard.

In order to compensate for this imbalance, SSCL generates significantly more instances than necessary and then selects them sensibly. A new parameter is intro-

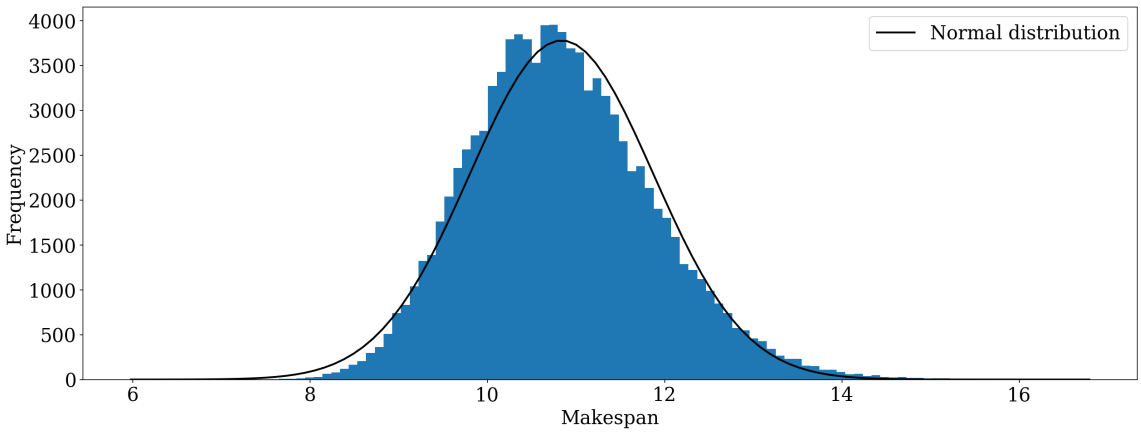

Figure 4.2: MTR Makespan distribution of 100,000 10×10 instances ($\mu \approx 11, \sigma \approx 1$)

duced for this purpose: *spacing* which describes how many more times the number of instances is generated. This procedure becomes clearer if one examines the sorted makespans as shown in Figure 4.3. The *spacing* factor now describes the size of unused instances, while the use segments are equally distanced. With a spacing of 8, this means that the unused areas are 7 (*spacing*-1) times as large as the used areas. This type of sampling means that both edges (most easy and hard instances) are always used, while the middle is only sparsely represented. As with CCL, the inverted ISSCL strategy is only characterised by the different sorting sequence.

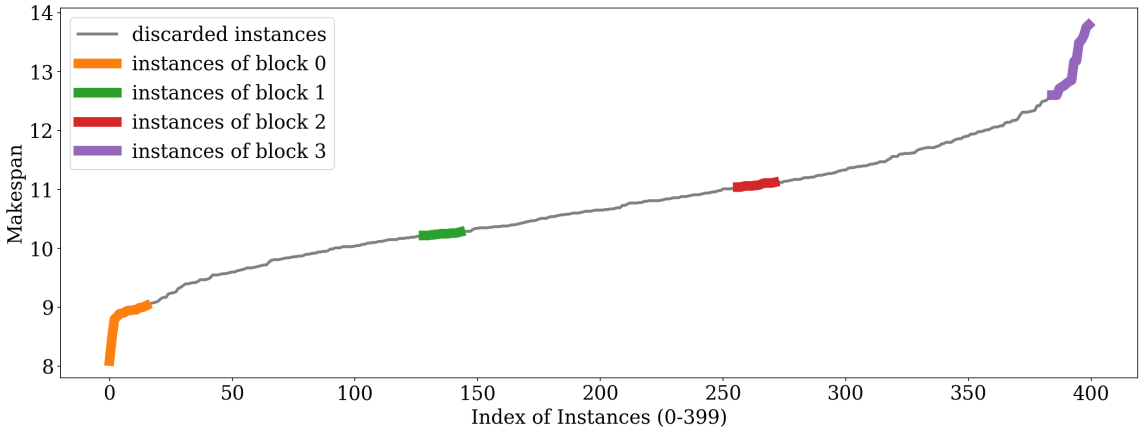

Figure 4.3: Visualisation of spaced sampling on sorted makespans with 4 levels, *spacing*=8 and *batch_size*=16

Figure 4.4 again shows the DTS distribution of the epochs. It can be seen that the first and last epochs of a period are shifted outwards to a greater extent than with CCL.

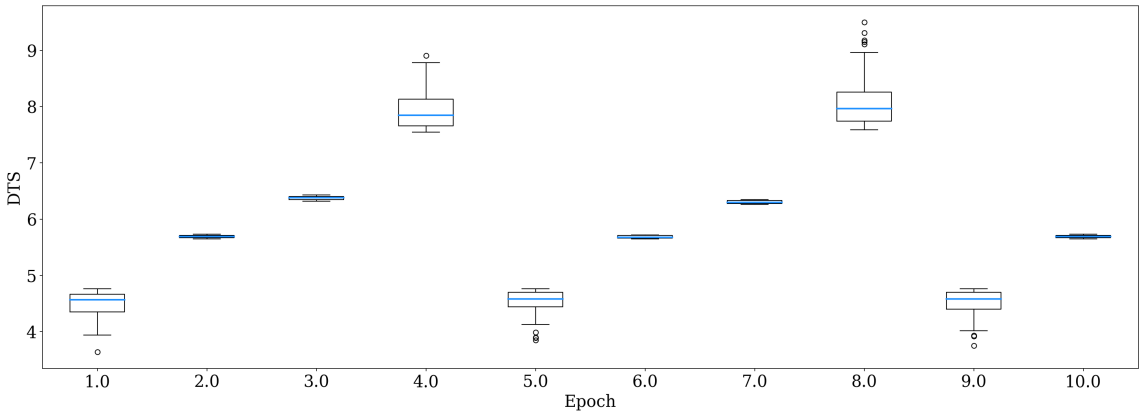

Figure 4.4: Visualisation of SSCLL with 4 levels, *factor*=8 and *batch_size*=128

Algorithm 5 Spaced Sampling Curriculum Learning (SSCL) algorithm

Inputs: initialised network parameters θ, array of *batchsizes*,
positive integer l_{min}, *stepsize*

$l \leftarrow l_{min}$
for $i = 1, \ldots, 3000$ **do**
  $g, l \leftarrow PDTC(i, l, \theta)$ ▷ see algorithm 1
  if $i\%stepsize == 0$ **then**
    $n \leftarrow batchsize[s] \cdot stepsize \cdot spacing - (spacing - 1) \cdot batchsize[s]$
    Sample current size s following distribution of g
    Generate n *instances* of size s
    Sort *instances* by *MTR* makespans ascending
    for $k = 0, \ldots, stepsize - 1$ **do**
      $start \leftarrow k \cdot batchsize[s] \cdot spacing$
      $end \leftarrow start + batchsize[s] \cdot spacing - (spacing - 1) \cdot batchsize[s]$
      $b[k] \leftarrow instances[start : end]$
    end for
  end if
  Update θ on $b[i\%stepsize]$
end for
return *trained parameters* θ

4.4 Hard Instances Curriculum Learning (HICL)

This section presents a strategy that tests a further assumption regarding the results of [7]. The authors of [7] came to the conclusion that it can be beneficial start training on the hardest instances. From this, the hypothesis can be derived that DRL agents generally train better with hard instances in this context. The *Hard Instances CL* (HICL) strategy is developed to investigate this hypothesis. It works in a similar way to SSCL, but with the difference that only the hardest instances

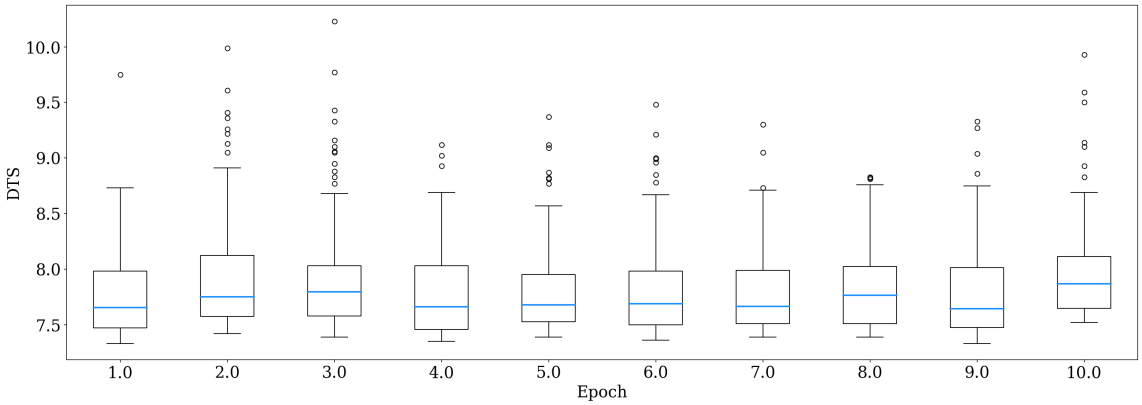

Figure 4.6: Visualisation of ISSCL with TODO parameter

are selected. Furthermore, there is also no more *stepsize*. The problem size is reselected for each epoch. The spacing factor determines how many more instances are generated than are required. Figure 4.5 visualises this type of sampling. The HICL algorithm is also shown below as pseudocode. Figure 4.4 again shows the DTS distribution of the epochs. All 10 epochs show an approximately equal distribution.

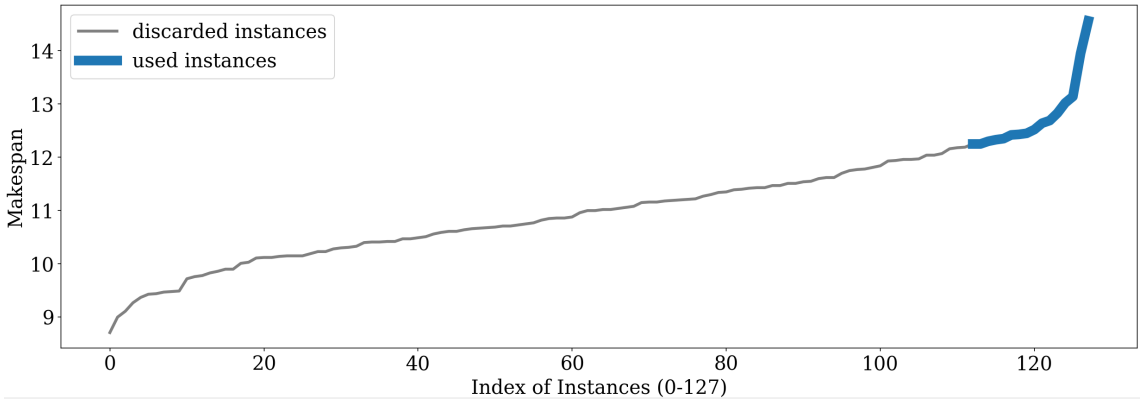

Figure 4.5: Sampling of only hard 6×6 instances with factor=8, batch_size=16

Algorithm 6 Hard Instances Curriculum Learning (HICL) algorithm

Inputs: initialised network parameters θ, array of *batchsizes*,
positive integer l_{min}, *factor*

$l \leftarrow l_{min}$
for $i = 1, \ldots, 3000$ **do**
  $g, l \leftarrow PDTC(i, l, \theta)$ ▷ see algorithm 1
  Generate $(batchsize[s] \cdot factor)$ *instances* of size s
  Sort *instances* by *MTR* makespans descending
  instances $\leftarrow$ *instances*[: $batchsize[s]$] first values of
  Update θ on *instances*
end for
return *trained parameters* θ

4.5 Interpolated Sizes Curriculum Learning (ISCL)

The *interpolated sizes curriculum learning* (ISCL) strategy takes the novel approach of expanding the space of problem sizes by using more than just the 10 SPS. While the other strategies interpret the set of problem sizes as a one-dimensional list and assign a selection probability to these sizes (denoted as g in algorithm 1), ISCL interprets the set of problem variables as a two-dimensional discrete space. If the 10 SPS are represented in this space (figure 4.7), the sparse population is quickly apparent. The assigned values in the figure are currently meaningless and have been equally distributed for visualisation purposes.

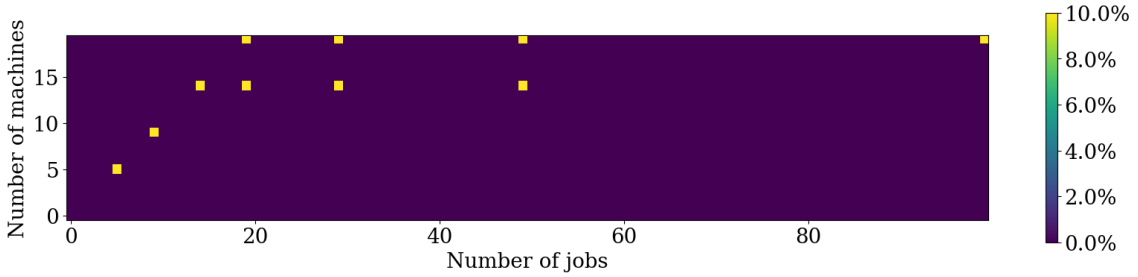

Figure 4.7: Two-dimensional discrete space of problem sizes with the 10 SPS

If the space of problem sizes is expanded in this way, two problems arise which must be solved. On the one hand, the probability distribution must now be calculated for many more sizes, and on the other hand, new rules must be created to generate a curriculum that can dynamically increase the difficulty based on the agent's performance. These problems are addressed in the following.

All RASCL-based CL strategies calculate the probability distribution of the problem sizes by calculating the gaps to a test data set with the 10 SPS. It takes about 90 seconds to determine these values for this sizes. If one wants to determine these values for all sizes considered in the space ($100 \cdot 20 = 2000$), this process would represent a large part of the entire training process. In order to shorten this process considerably, ISCL only uses the gaps of the 10 SPS and interpolates all missing values linearly, as described in Algorithm 7.

In order to define a curriculum which, similar to RASCL, gradually incorporates more problem variables over time, two dimensions must now be considered and no longer just a linear one-dimensional list. The RASCL level l is also determined for this purpose. This level represents a problem size with n jobs and m machines,

i.e. $n \cdot m$ operations in total. This number of operations is regarded as a limit in ISCL and represents a dynamic limit. In addition, 4 further rules are defined in the following in order to limit the sizes to those similar to the SPS:

Limit	**Definition**	**Description**	**Type**
$(l1)$	$n \cdot m \leq n_l \cdot m_l \cdot 1.1$	"not more operations than 110% of the operations of the current level l"	dynamic
$(l2)$	$m \geq 3$	"at least 3 machines"	static
$(l3)$	$n \geq m$	"at least as many jobs as machines"	static
$(l4)$	$m \geq \frac{n}{5}$	"not more than 5 jobs per machine"	static
$(l5)$	$m \leq 20$	"not more than 20 machines"	static

Table 4.3: Definition of the ISCL space contraints

Figure 4.8 visualises these 5 limits in a continuous space. n_l and m_l represent the number of jobs and machines of the current RASCL level l. Here, 30×20 is chosen as an example for l These contraints can also be discretised in order to use them to select valid JSSP sizes.

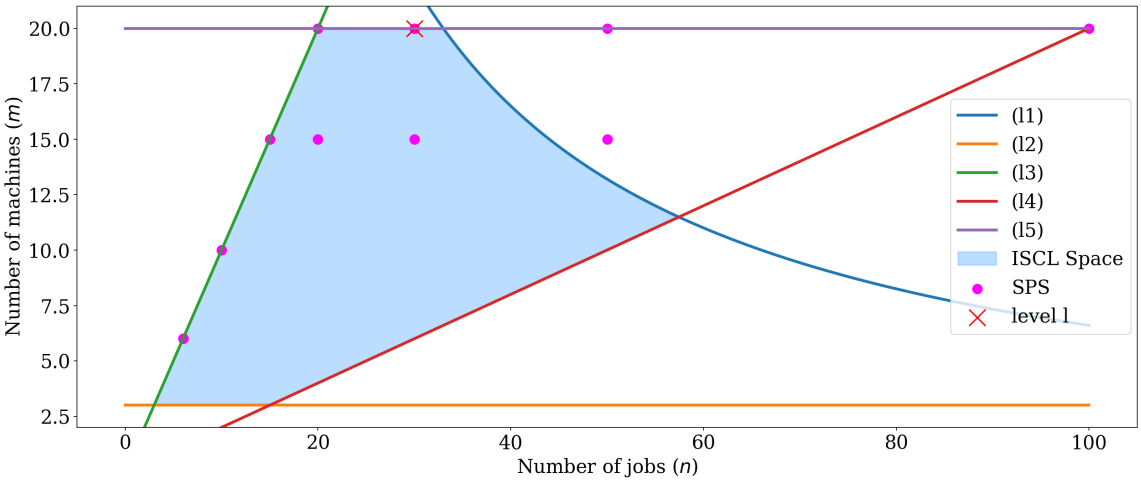

Figure 4.8: Visualisation of the ISCL space constraints

The following figure 4.9 shows the calculated probability distributions (denoted as a *map* in algorithm 7) at two different points in time: after 150 and 1200 epochs. It shows both how the dynamic limit works and a shift in the probabilities towards larger problem sizes. This indicates that the agent can already solve the smallest sizes better, which leads to a smaller gap.

Warning: The *PDTC()* method limits the distribution and applies softmax in lines 13 and 14. However, the ISCL strategy requires the distribution without these two transformations, and assumes that these transformations are not carried out.

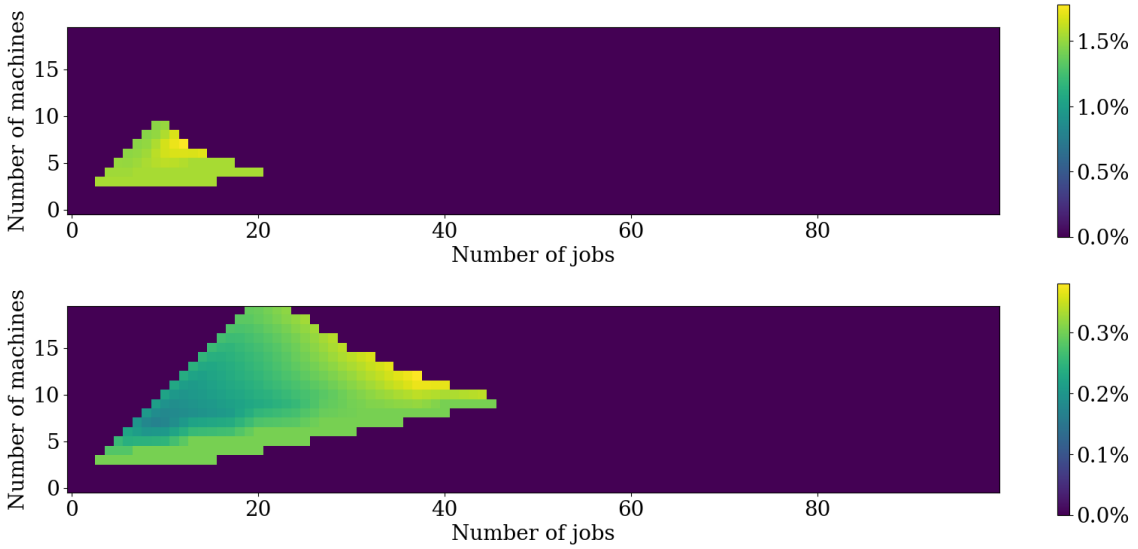

Figure 4.9: ISCL *map* after 150 and 1200 epochs of training

Algorithm 7 Interpolated Sizes Curriculum Learning (ISCL) algorithm

Inputs: initialised network parameters θ, array of *batchsizes*, positive integer l_{min}

1: $l \leftarrow l_{min}$
2: **for** $i = 1, \ldots, 3000$ **do**
3: $g, l \leftarrow PDTC(i, l, \theta)$ ▷ see algorithm 1 and **warning**
4: $map \leftarrow 100 \times 20$ 2d-array filled with of 0
5: **Fill** *map* with values of g
6: **Interpolate** every 0 of *map* linearly
7: **Blur** *map* (average of neighbouring values)
8: $map \leftarrow \text{softmax}(map)$
9: $n, m \leftarrow$ size of l
10: $limit \leftarrow n \times m \times 1.1$
11: **for** $x = 1, \ldots, 100$ **do**
12: **for** $y = 1, \ldots, 20$ **do**
13: **if** $x \cdot y > limit$ **or** $y > x$ **or** $y < 3$ **or** $x \backslash y > 5$ **then**
14: $map[x][y] \leftarrow 0$
15: **end if**
16: **end for**
17: **end for**
18: $map \leftarrow map \setminus \text{sum}(map)$
19: **Sample current size** s following distribution of *map*
20: **Generate** $(batchsizes[s])$ *instances* of size s
21: **Update** θ on $b[i\%stepsize]$
22: **end for**
23: **return** *trained parameters* θ

Chapter 5

Experiments

5.1 Experimental Setup

Three agents per curriculum learning strategy are trained. All hyperparameters are fixed for all tests. The values used for the batch size, learning rates and other parameters can be found in the appendix. All metrics are logged using the developer platform "Weights & Biases" [44]. Not only the metrics from 5.2 are recorded, but also other basic measured values. All experiments were carried out on a Linux server with a 16-core CPU and 256GB RAM.
A total of 4 different sets of JSSP instances are used during the experiments. To provide an overview of these, they are all listed here.

(TA) Taillard's set [33] (as in 2.1.3) with 80 instances for the calculation of the *gap difference* (following section) and the final results (section 5.3.5)

($D1$) 100 instances (10 per SPS) for calculating the performance distribution g in algorithm 1 (referred to as *testdata*)

($D2$) 80 instances (10 per TA size) for calculation of the *Relative Gap* as in 5.2

($D3$) Training data that gets randomly generated on the fly as described in the algorithms

To increase the statistical significance, all experiments are carried out three times with different seeds. A total of 24 agents are trained. Three agents each are trained with RASCL and NOCL as a baseline and for comparison. Subsequently, three

agents are trained with each of the following CL strategies: CCL, ICCL, SSCL, ISSCL, HICL, ISCL. The appendix contains the parameters used for the strategies.

5.1.1 Used Inference Strategies

A total of three different inference strategies are used during the experiments. The table 5.1 provides an overview of the different procedures. It is important to note that there is a change during the experiments. It makes sense to use greedy for calculating the performance distribution g, so that the agents make optimal decisions during testing (exploitation). However, the first eight agents were tested using sampling with as *sizesearch* of 1. The following 16 agents used greedy for this calculations. This means that a third of the agents were not tested optimally. However, the temporal limitations of this work do not allow the repeated training of 8 additional agents. Nevertheless, the results in the following sections do not show any significant difference between these two groups of agents.

Procedure	**Dataset**	**Inference Strategy**	**Parameter**
Training	D3	sampling	*sizesearch*=1
Performance Distribution (g)	D1	greedy	-
Gap Difference	TA	sampling	*sizesearch*=128
Relative Gap	D2	greedy	-
Final Results	TA	sampling	*sizesearch*=128

Table 5.1: Use of different inference strategies for each procedure of the experiments

5.2 Custom Metrics / Performance Measurement

This section addresses the challenge of measuring the performance of the DRL models during and after the training. As mentioned in Section 2.1.1, the optimality gap is the key indicator for classifying how well a model can solve a JSSP. In order to be able to calculate this value, the optimal makespans of the instances must be known, as is the case with the set from Taillard [33]. Therefore, after the training process, the final results of a model are determined by solving the TA instances and calculating the optimality gaps.

Measuring the performance of a model during training could be done in a similar

way, but then the frequent testing in between training would take up a large part of the time. This calculation for all 80 TA instances takes about 90 minutes. For this reason, a trade-off is made. These exact values are only determined every 500 epochs (about 4 times in total).

Another metric can now be easily derived from these optimality gaps. Since the best optimality gaps are known, which were achieved on the Taillard set in [6], the difference between these values can now be regarded as a further metric, called *gap difference* (GD), which is recorded every $rate_{GD} = 500$ epochs. This value describes the difference between the equally weighted average of all optimality gaps of RASCL on the TA set and the equally weighted average of all optimality gaps of the currently trained model on the TA set. This metric represents the difference in percentage points and starts in the negative range at the beginning of the training process and rises above 0 in the best case. It is a simple metric to recognise how well a model performs compared to the results of [6], while neglecting how well it performs on a specific problem size.

In order to investigate not only the final performance, but also the learning behaviour, a metric is required which can also evaluate the performance during the training process faster than mentioned above. A faster metric to estimate the performance of a model during training uses a comparison to an already trained model, called *relative gap* (RG). For this purpose, another data set (D2) is randomly generated and solved with a pre-trained model, resulting in a set of makespans that are sufficient but most likely not optimal. The model was trained with RASCL up to a JSSP problem size of 20×20. This pre-calculation only needs to be done once before training all models. The data set generated for this purpose contains the 10 SPS. In order to decide how many instances a problem size should comprise, one must weigh up between statistical significance and calculation speed. One instance per size would lead to a very fast calculation, but would also be heavily dependent on the random initialisation. A higher number reverses both effects accordingly. In this work, 10 instances per size are chosen in order to find a balance. It should be noted that this metric depends not only on the random initialisation of the instances, but also strongly on the pre-trained model. The metric itself is characterised by the gap to the pre-calculated makespan. It uses the calculation as one optimality gap (equation 2.1), but replaces C^*_{max} by the makespan of the pre-trained model. This

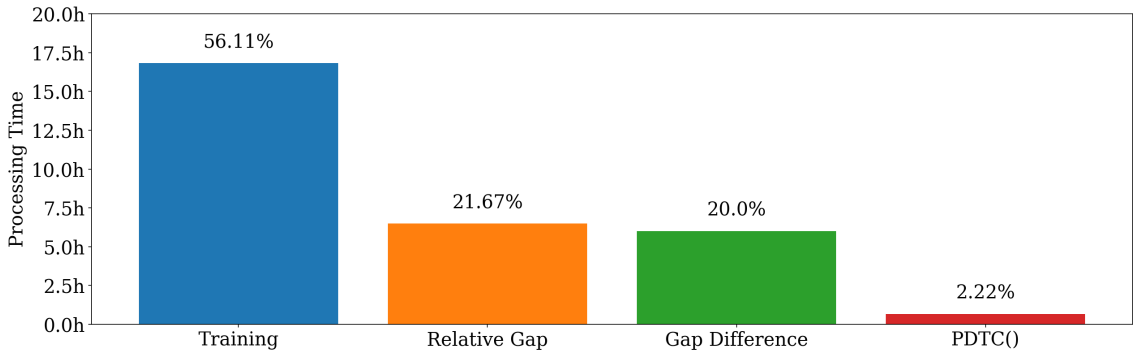

Figure 5.1: Distribution of processing time spent on all training procedures by one agent

relative gap is suitable for a quick and relative comparison between different CL strategies. The absolute number, however, is less meaningful as it only represents the comparison to a freely chosen model. The calculation of this metric takes about 30 minutes and is recorded every $rate_{RG} = 150$ epochs. Figure 5.1 shows the distribution of processing times for all procedures. The relative gap is used to determine when a model is saved. If a new lowest value is reached during training, the model is saved. Saving the models is important for the final tests. The two metrics together require over 40% of the total training time for a DRL agent.

5.3 Experiment Results

This section presents the test results, beginning with a cautionary note regarding a potential implementation error. The results are then categorised and presented.

5.3.1 Disclaimer

The first important observation to make is that all agents (including the one trained with RASCL) achieve at least 13%p worse results than in [6].

It is difficult to find the reasons for this difference. The same inference strategy was used and for RASCL, an equivalent implementation was used. There is only one difference between the setup of [6] and these experiments. In this work, the batch sizes were halved due to limited computational capacity. However, training with the same batch sizes only improves the results by 0.041%p. This adjustment therefore does not close this large gap.

Thus, an error in the implementation of this work cannot be ruled out. Since

no other possible reason is found, all further results and discussions of these refer to the relative differences between the CL strategies, which can still be determined. The baseline comparison is therefore not an agent from [6], but an agents trained with RASCL in this work.

5.3.2 Gap Difference

The *Gap Difference* metric is visualised below in figure 5.2. There are 24 points every 500 epochs, which represent the respective agents. Furthermore, for each of the 8 strategies, an average of the three values per point in time is shown as a line. As described in section 5.2, it shows the average difference in percentage points to the best results achieved with RASCL in [6], which provides a quick overview of how well an agent performs overall. The values at step 0 are not shown here, as these only depend on the random initialisation and do not depend on the agent, but would change the scaling considerably.

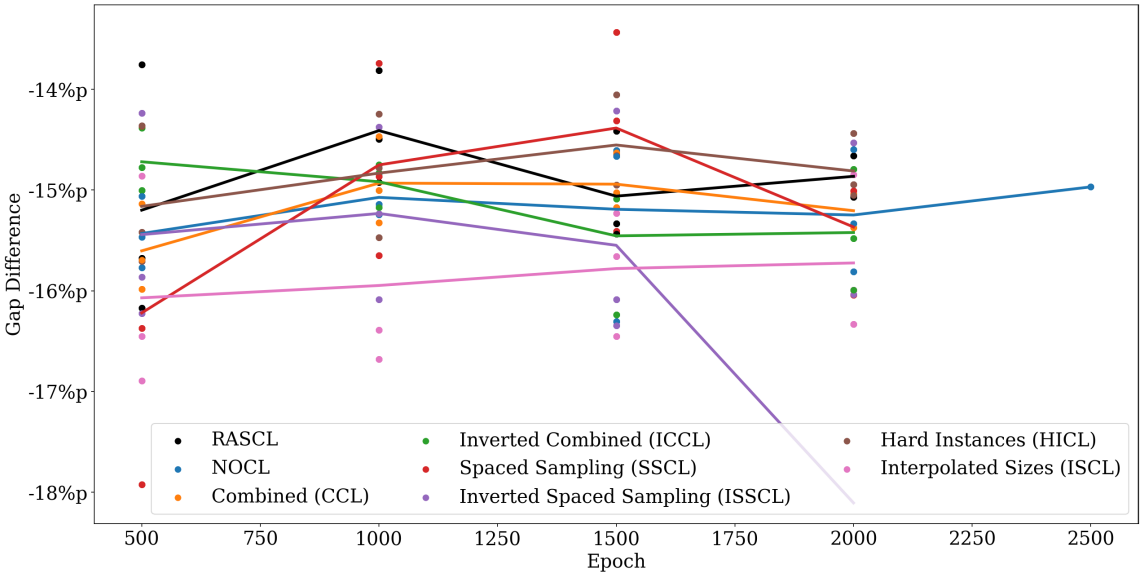

Figure 5.2: Gap Differences (higher is better)

It should be noted that one data point is omitted to make the figure easier to read: the ISSCL strategy has reached a GD value of approximately -23%p at 2000 epochs.

Since this metric only produces the first meaningful value after 500 training steps, it is not possible to recognise a nice learning progression for all CL strategies at the beginning, as the RG does. It is immediately noticeable that NOCL has one measuring point more than all other strategies. This is due to the fact that NOCL

achieves the worst results in general, as well as the fact that this agent trains the slowest and finishes about 500 training steps later. This behaviour is to be expected and not unreasonable, since CL can be effective in many areas, as already mentioned in 2.3.

Most of the strategies achieve GDs of approximately -17%p to -13%p and are therefore relatively close to each other. The best GD is achieved by the SSCL strategy with a value of -13.43%p, closely followed by RASCL with $-13.75\%p$.

The results of this metric are very noisy overall and make it difficult to clearly rank the strategies. Contrary to previous assumptions, the agents which train without a curriculum (NOCL) are not the worst. This contradicts the previous assumptions and the referenced results.

Apart from the one strongly deviating point from ISSCL (which is not shown), the ISCL strategy is clearly the worst on average. There could be several reasons for this. It could be due to the fact that the agents train many JSSP sizes that are not covered by this metric. A large part of the time is therefore invested in training variables that are not part of the SPS and are therefore not tested by this metric. Furthermore, the ISCL space is spatially expanded in a direction in which more jobs and fewer machines are represented. This can be clearly seen in Figure 4.8, which shows the blue area, which tends to be extended downwards and to the right. It is possible that an extension in another or several directions would improve the results.

The second worst performing strategy is ISSCL. This result is particularly interesting because the non-inverted variant (SSCL) achieves better results. This observation suggests that, contrary to the results of Waubert de Puiseau et al. [7], it is helpful to start training with easy instances.

Overall, it can also be observed that most strategies are difficult to separate from one another according to this metric. RASCL, SSCL, HICL, CCL, NOCL and ICCL are very close to each other and often cross during training. This proximity (also to NOCL) would indicate a low relevance of a CL strategy in this context.

5.3.3 Relative Gap

Figure 5.3 shows the relative gap metric in the same way as for the gap difference. In this figure, the first data point of each strategy is also removed, as this has no

significance for the strategy and changes the scaling considerably. It is important to note that, as can be seen from the truncated line, the scaling of the y-axis has been greatly adjusted, as an agent trained with the HICL strategy reaches an RG value of approximately 250% at one point. Since the exact value does not play a major role, the scaling is adjusted so that all other values are easier to recognise. Two points of an agent trained with ISSCL are also outside the selected scaling. These are located at around 25% and can be easily recognised by the upward deflections of the average line.

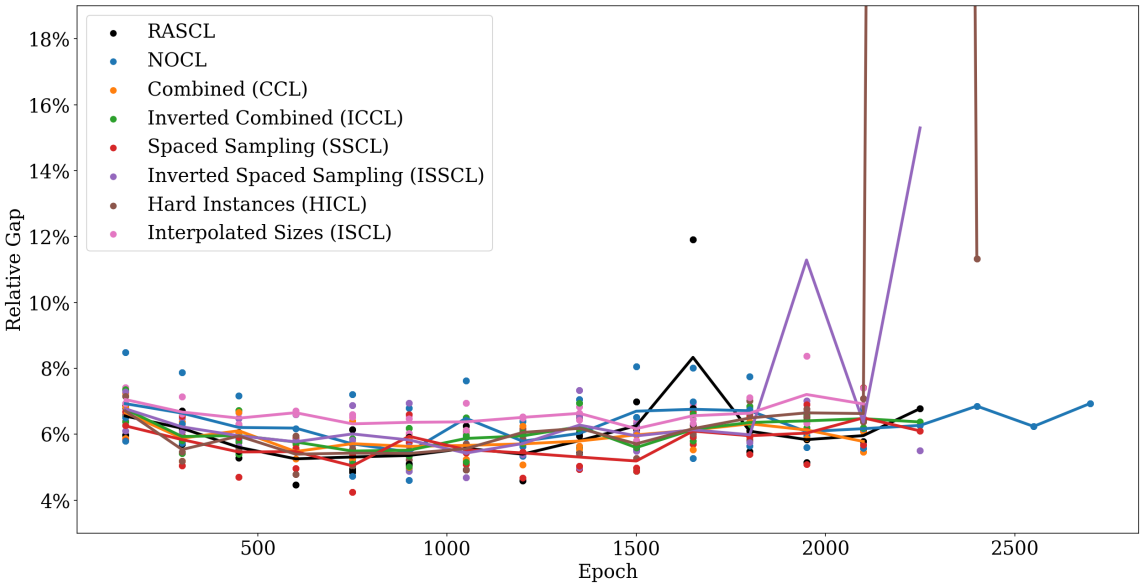

Figure 5.3: Relative Gaps (lower is better)

These large deviations can also be found to a similar extent in the achieved reward. Both agents receive correspondingly lower rewards at these points in time. These deviations can presumably be attributed to the generally unstable learning behaviour of DRL agents. Such a large difference is unlikely to be due to the differences between the various strategies.

This metric shows largely the same results as the gap difference. Agents trained with ISCL perform the worst and ISSCL also leads to a poor result. NOCL is also represented roughly in the middle again. In contrast to GD, a training process is shown at the beginning up to around 1000 epochs, as the sampling rate is significantly higher for this metric. Particularly in the first 500 epochs, a clear reduction can be seen for all strategies. The observable correlation between these metrics is discussed in more detail in the following section.

5.3.4 Correlation between Relative Gap and Gap Difference

This section addresses the relationship between the two metrics *relative gap* (RG) and *gap difference* (GD). Both metrics are intended to measure how well an agent is performing. RG measures the difference to a pre-trained model and GD measures the performance on a known data set. Under these circumstances, one would expect a correlation that if one metric improves, the other will also improve. Note that RG improves when the value decreases and GD improves when the value increases, as described in 5.2.

Figure 5.4 shows a comparison of the measured values in a scatter diagram. The GD is shown on the X-axis and the RG on the Y-axis. Each strategy is represented by 12 measurement points at the time points 500, 1000, 1500, 2000. As the GD is measured every 500 steps and the RG every 150 steps, there are only a few points in time at which both metrics are recorded. For this reason, the closest measurement point is used, which could slightly distort the presentation. The graph includes a linear regression to facilitate the identification of the correlation.

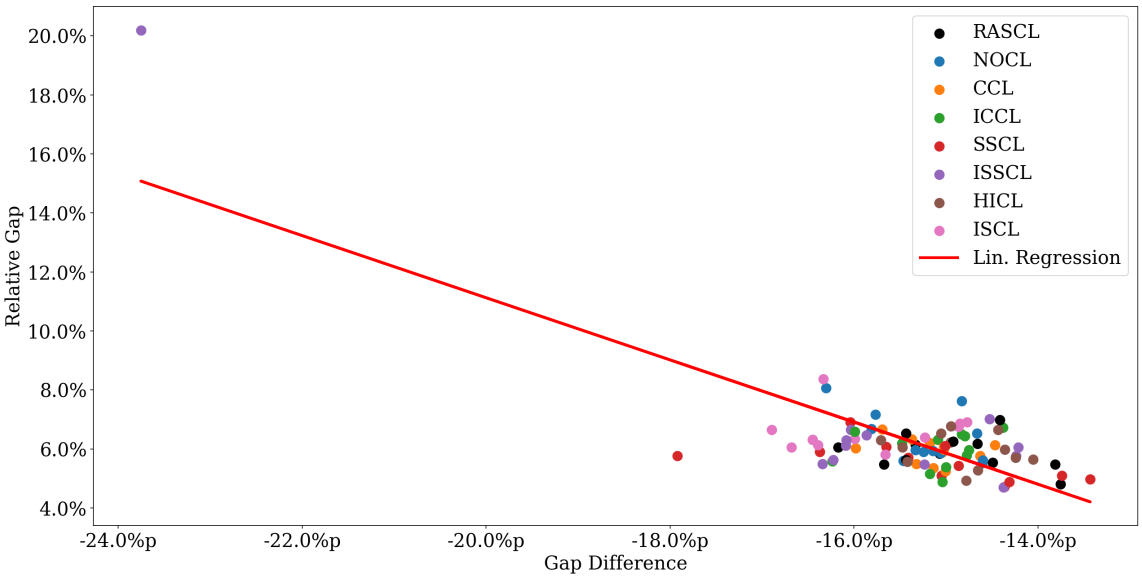

Figure 5.4: Correlation between *Relative Gap* and *Gap Difference*

The *Pearson Correlation Coefficient* (PCC) can be used to quantify the correlation between these metrics. The PCC measures the strength and direction of the linear relationship between two variables. The value can be between -1 and 1, where -1 represents a perfect negative linear relationship, 1 a perfect positive linear relationship and 0 no linear relationship. It is the covariance of the two variables divided

by the product of their standard deviations, as defined in equation 5.1. Here, X describes the gap differences and Y the relative gaps.

$$\rho_{X,Y} = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y} \tag{5.1}$$

In this case, the PCC takes on a value of $\rho \approx -0.7697$. This negative value indicates that there is a moderate to strong negative linear relationship between the two variables. The correlation is very significant with a p-value of about $10^{-17}\%$, indicating that the current result would be very unlikely if the correlation coefficient were zero. This result demonstrates that the two metrics exhibit similar behaviour. If a model improves in one metric, it also tends to achieve better results in the other metric.

5.3.5 Results on Taillard's instances

The following results on the instances of Taillard [33] have the highest significance, as they use the models from a point in time at which they were best and are determined with the best inference strategy and additionally.

The figure 5.5 consists of 8 groups with 8 bars with error bars representing the corresponding minimum and maximum values of the three experiments per CL-strategy. The bars represent the average optimality gap over the three experiments per strategy. Furthermore, the results of [6] are also shown as a dashed line.

Apart from the major difference to [6] mentioned in the disclaimer, small regular differences between the CL strategies can be identified. For the 8 problem sizes of the TA set analysed, HICL is ranked first four times and second four times, making it the best CL strategy overall according to these results. However, it should be noted that these differences are on the one hand very small and on the other hand not so significant, as large error bars indicate a strong variance.

As the differences between the strategies are difficult to recognise with this scaling, a performance order for the strategies is created for each problem size. From this, an average can be determined for each strategy, resulting in an easy-to-read ranking of the strategies as shown in Figure 5.6.

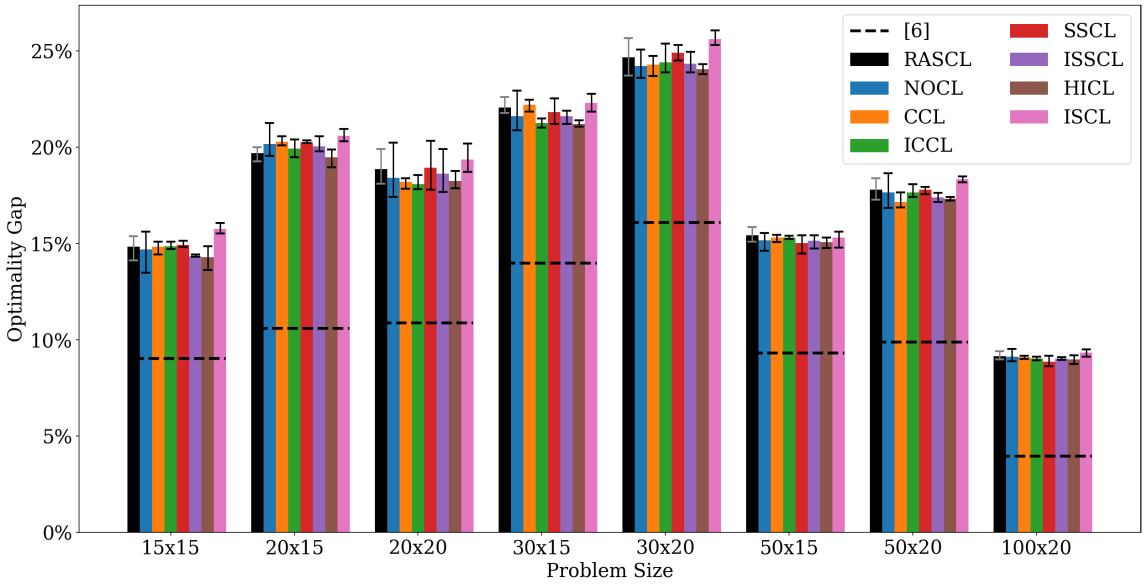

Figure 5.5: Final Results on TA with sampling (*sizeserach*=128). Order of strategies from left to right: RASCL, NOCL, CCL, ICCL, SSCL, ISSCL, HICL, ISCL

Figure 5.6 clearly shows HICL as the the best CL strategy, while ISCL is almost always the worst. Furthermore, based on this ranking, every strategy developed in this thesis except ISCL is better than RASCL. Another interesting observation is that the two inverted variants are better than the non-inverted ones: ICCL is better than CCL and ISSCL is better than SSCL. This also fits with HICL being the best, as the inverted strategies and HICL all start with the most difficult instances. This thus confirms the hypothesis from Section 4.4 that DRL agents achieve better final results when they start training on difficult JSSP instances. However, this figure must be viewed in its entire context, as it may greatly emphasise the differences.

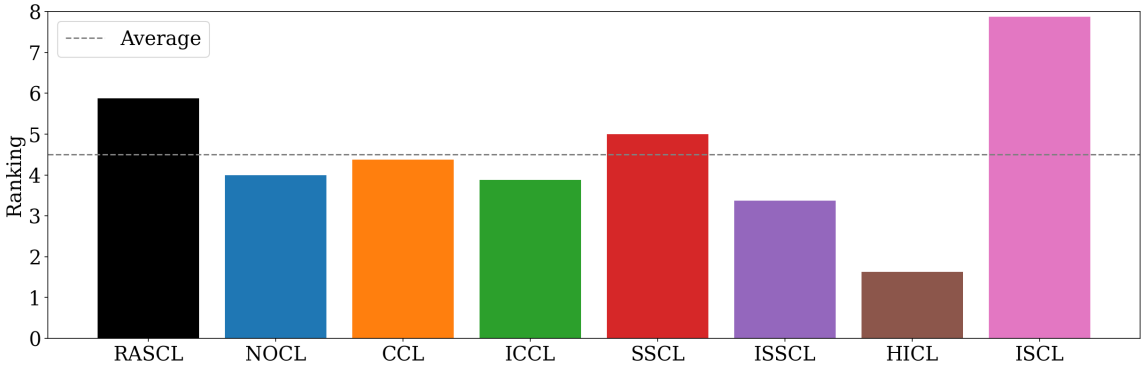

Figure 5.6: Average ranking of strategies based on figure 5.5 (lower is better)

5.4 Summary of Results

This section summarises the results of the various CL approaches. It presents the most interesting observations, but does not go into full detail.

Firstly, the disclaimer in Section 5.3.1 should be reiterated at this point, as it was not possible to reproduce the results of [6] on Taillard's instances [33] within the scope of this work. The best DRL agents of this work achieve optimality gaps which are about 138% (+13%p) worse than those of Iklassov et al. [6]. This could be due to an undetected implementation error in this work. For this reason, the strategies are not compared with the results of [6], but are analysed in a relative comparison to each other. Overall, a strong variance in the results can be observed, so that the range of optimality gaps overlaps strongly between the strategies. Nevertheless, consistent differences between the strategies can be observed on average.

The CL strategy that leads to the worst results is ISCL. This strategy expands the space of problem sizes used for training. It is the only one that is worse than the comparison strategy RASCL. There could be two possible reasons for this. Either the space of problem sizes has been chosen unfavourably or the measurement, which is limited to 8 sizes, cannot fully and correctly capture the performance of the agents.

The strategies CCL, ICCL, SSCL and ISSCL are all slightly better than RASCL but are subject to relatively large fluctuations in the results. This class of strategies uses a second dimension so that both the problem size and the difficulty of the instances within problem size are selected specifically and dynamically. Nevertheless, a very interesting observation can be made within this class of strategies: The strategies that start with the most difficult instances (ICCL and ISSCL) are each better than the variants that do exactly the opposite (CCL and SSCL). These results are consistent with [7].

The best results are achieved by the HICL strategy. A DRL agent that uses this strategy for training only uses the most difficult instances. This is therefore a continuation of the results of [6]. This strategy achieves the lowest optimality gaps in almost every problem size and is only dependent on a single parameter (*spacing*).

Another interesting aspect is the fact that the use of no curriculum (NOCL) does not always and clearly lead to a worse result than RASCL. The results from Figures 5.5 and 5.6 even show the opposite: on average, NOCL is better than RASCL. If this

result is not due to an error in the implementation of this work, it could be because Iklassov et al. [6] did not train their models without a curriculum for comparison. They used other methods for comparison. Thus, the developed model itself could already lead to an advantage over other methods, independent of a CL approach.

Chapter 6

Future Work

After summarising the results of this work from the last chapter, this chapter deals with potential future work.

The development and investigation of an inverted variant of RASCL could yield interesting results. So far, the assumption has always been that DRL agents learn best when they start on small JSSP problem sizes and increase them during training. The fact that the reverse order has not yet been investigated and that [7] and the results of this work show that it is advantageous to start with hard instances (within a problem size) strengthens the hypothesis that RASCL inverted could be better than RASCL. A confirmation of this hypothesis would then also lead to the assumption that the other CL strategies presented here could also be better when inverted.

The modification and extension of the performance measurements of the agents could be the subject of further work. The measurements in this paper are limited to the 10 SPS. While ISCL is the only strategy that extends the space of problem sizes for training, it is still only evaluated on the 10 SPS or the 8 TA sizes. The investigation and evaluation of all strategies on an extended set of problem sizes could yield further effects and results. Furthermore, the space of problem sizes could generally be expanded differently than is the case with ISCL.

The use of MTR to determine the difficulty of an instance has proven to be useful, but other methods to determine this could achieve even better results. Finally, a more frequent repetition of all experiments would further increase the statistical significance of the results, as this work used only 3 repetitions.

References

[1] T. Yamada and R. Nakano, *Genetic algorithms in engineering systems*. No. 55 in IEE control engineering series, London: Institution of Electrical Engineers, 1997.

[2] H. Xiong, S. Shi, D. Ren, and J. Hu, "A survey of job shop scheduling problem: The types and models," *Computers & Operations Research*, vol. 142, p. 105731, June 2022.

[3] D. Hoitomt, P. Luh, and K. Pattipati, "A practical approach to job-shop scheduling problems," *IEEE Transactions on Robotics and Automation*, vol. 9, pp. 1–13, Feb. 1993.

[4] B. Yan, M. A. Bragin, and P. B. Luh, "Novel Formulation and Resolution of Job-Shop Scheduling Problems," *IEEE Robotics and Automation Letters*, vol. 3, pp. 3387–3393, Oct. 2018.

[5] L. Perron and V. Furnon, "Or-tools." `https://developers.google.com/optimization/`.

[6] Z. Iklassov, D. Medvedev, R. Solozabal, and M. Takac, "Learning to generalize Dispatching rules on the Job Shop Scheduling," Nov. 2022. arXiv:2206.04423 [cs].

[7] C. Waubert De Puiseau, H. Tercan, and T. Meisen, "Curriculum Learning in Job Shop Scheduling using Reinforcement Learning," 2023. Publisher: Hannover : publish-Ing.

[8] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 1 ed., Apr. 2009.

[9] A. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present and future," *European Journal of Operational Research*, vol. 113, pp. 390–434, Mar. 1999.

[10] O. Shylo, "Best known lower and upper bounds: Job Shop Scheduling Problem : Taillard's instances.," Dec. 2020.

[11] O. Shylo, "Best known lower and upper bounds: Job Shop Scheduling Problem : Demirkol's instances.," Dec. 2020.

[12] M. Kemmerling, M. Combrzynski-Nogala, A. Gannouni, A. Abdelrazeq, and R. H. Schmitt, "Job and Operation Entropy in Job Shop Scheduling: A Dataset," Aug. 2023.

[13] S. S. Panwalkar and W. Iskander, "A survey of scheduling rules," *Operations research*, vol. 25, no. 1, pp. 45–61, 1977.

[14] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management science*, vol. 34, no. 3, pp. 391–401, 1988.

[15] R. Cheng, M. Gen, and Y. Tsujimura, "A tutorial survey of job-shop scheduling problems using genetic algorithms—i. representation," *Computers & industrial engineering*, vol. 30, no. 4, pp. 983–997, 1996.

[16] M. Dell'Amico and M. Trubian, "Applying tabu search to the job-shop scheduling problem," *Annals of Operations research*, vol. 41, no. 3, pp. 231–252, 1993.

[17] E. F. Morales and H. J. Escalante, "A brief introduction to supervised, unsupervised, and reinforcement learning," in *Biosignal Processing and Classification Using Computational Learning and Intelligence*, pp. 111–129, Elsevier, 2022.

[18] Y. Li, "Deep Reinforcement Learning: An Overview," Nov. 2018. arXiv:1701.07274 [cs].

[19] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, Sept. 2013.

[20] D. Vengerov, "A reinforcement learning approach to dynamic resource allocation," *Engineering Applications of Artificial Intelligence*, vol. 20, pp. 383–390, Apr. 2007.

[21] Y. Bi, S. Kapoor, and R. Bhatia, eds., *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016: Volume 2*, vol. 16 of *Lecture Notes in Networks and Systems*. Cham: Springer International Publishing, 2018.

[22] P. Soviany, R. T. Ionescu, P. Rota, and N. Sebe, "Curriculum Learning: A Survey," *International Journal of Computer Vision*, vol. 130, pp. 1526–1565, June 2022.

[23] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, 2009.

[24] L. Jiang, D. Meng, Q. Zhao, S. Shan, and A. Hauptmann, "Self-Paced Curriculum Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, Feb. 2015.

[25] M. Lisicki, A. Afkanpour, and G. W. Taylor, "Evaluating Curriculum Learning Strategies in Neural Combinatorial Optimization," Nov. 2020. arXiv:2011.06188 [cs].

[26] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997. Conference Name: Neural Computation.

[27] O. Vinyals, S. Bengio, and M. Kudlur, "Order Matters: Sequence to sequence for sets," Feb. 2016. arXiv:1511.06391 [cs, stat].

[28] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural Combinatorial Optimization with Reinforcement Learning," Jan. 2017. arXiv:1611.09940 [cs, stat].

[29] W. Kool, H. van Hoof, and M. Welling, "Attention, Learn to Solve Routing Problems!," Feb. 2019. arXiv:1803.08475 [cs, stat].

[30] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min, "POMO: Policy Optimization with Multiple Optima for Reinforcement Learning," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 21188–21198, Curran Associates, Inc., 2020.

[31] C. K. Joshi, T. Laurent, and X. Bresson, "An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem," Oct. 2019. arXiv:1906.01227 [cs, stat].

[32] L. Wang, X. Hu, Y. Wang, S. Xu, S. Ma, K. Yang, Z. Liu, and W. Wang, "Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning," *Computer Networks*, vol. 190, p. 107969, May 2021.

[33] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, pp. 278–285, Jan. 1993.

[34] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi, "Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 1621–1632, Curran Associates, Inc., 2020.

[35] Google LLC, "Google scholar." `https://scholar.google.de/`, Jan. 2024.

[36] F. M. Wittmann, "WittmannF/sort-google-scholar," Jan. 2024. original-date: 2016-04-30T00:55:03Z.

[37] W. Zhang and T. G. Dietterich, "A reinforcement learning approach to job-shop scheduling," in *Ijcai*, vol. 95, pp. 1114–1120, 1995.

[38] M. E. Aydin and E. Öztemel, "Dynamic job-shop scheduling using reinforcement learning agents," *Robotics and Autonomous Systems*, vol. 33, no. 2-3, pp. 169–178, 2000.

[39] B. Waschneck, A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp, and A. Kyek, "Optimization of global production scheduling with deep reinforcement learning," *Procedia Cirp*, vol. 72, pp. 1264–1269, 2018.

[40] S. Luo, "Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning," *Applied Soft Computing*, vol. 91, p. 106208, 2020.

[41] J. Shahrabi, M. A. Adibi, and M. Mahootchi, "A reinforcement learning approach to parameter estimation in dynamic job shop scheduling," *Computers & Industrial Engineering*, vol. 110, pp. 75–82, 2017.

[42] Python Software Foundation, "Slice objects - python 3.9.17 documentation." `https://docs.python.org/3.9/c-api/slice.html`, Jan. 2023.

[43] J. Przywóski, "[] (slicing) - python reference (the right way) 0.1 documentation." `https://python-reference.readthedocs.io/en/latest/docs/brackets/slicing.html`, July 2015.

[44] Weights and Biases Inc., "Weights & biases: The ai developer platform." `https://wandb.ai/site`, Jan. 2024.

Appendix

Final Results on Taillard's Instances

1. Run	15 × 15	20 × 15	20 × 20	30 × 15	30 × 20	50 × 15	50 × 20	100 × 20
RASCL	15.38	19.78	18.59	21.78	24.6	15.31	17.73	9.05
NOCL	15.62	21.25	20.24	22.93	25.06	15.55	18.64	9.52
CCL	14.42	20.57	18.39	22.32	24.41	15.44	**16.94**	9.0
ICCL	14.74	20.4	18.54	21.49	25.37	15.39	18.07	9.12
SSCL	14.8	20.35	18.71	22.54	25.3	15.11	17.94	9.16
ISSCL	**14.3**	19.81	**17.67**	21.2	**23.88**	**14.75**	17.16	9.1
HICL	14.34	**18.96**	17.85	**21.18**	24.07	14.76	17.34	**8.75**
ISCL	15.52	20.95	19.19	22.77	25.5	14.79	18.17	9.12
2. Run	15 × 15	20 × 15	20 × 20	30 × 15	30 × 20	50 × 15	50 × 20	100 × 20
RASCL	14.13	**19.27**	18.11	21.81	**23.72**	15.1	17.27	8.97
NOCL	**13.47**	19.69	**17.42**	**21.05**	23.98	14.62	**16.84**	8.97
CCL	15.01	20.1	17.85	21.85	24.73	15.41	16.86	9.17
ICCL	15.08	19.9	17.9	21.22	23.89	15.27	17.4	9.03
SSCL	14.8	20.24	17.78	21.69	24.88	**14.49**	17.85	**8.62**
ISSCL	14.43	20.56	19.9	21.75	24.94	15.25	17.63	8.95
HICL	13.63	19.59	18.13	21.4	23.8	15.15	17.4	9.19
ISCL	16.06	20.54	20.19	22.29	26.06	15.53	18.49	9.4
3. Run	15 × 15	20 × 15	20 × 20	30 × 15	30 × 20	50 × 15	50 × 20	100 × 20
RASCL	15.01	19.99	19.91	22.6	25.65	15.86	18.38	9.41
NOCL	15.01	19.55	17.55	20.87	23.61	15.34	17.46	8.88
CCL	15.08	20.14	18.37	22.47	23.69	15.07	17.66	9.14
ICCL	14.71	19.46	17.81	21.02	23.96	15.24	17.5	8.94
SSCL	15.15	20.21	20.33	21.21	24.5	15.44	17.57	8.8
ISSCL	14.31	19.79	18.29	21.89	24.19	15.42	17.35	9.02
HICL	14.86	19.87	18.76	21.07	24.31	15.31	17.23	8.99
ISCL	15.71	20.29	18.72	21.83	25.3	15.61	18.35	9.5
[6]	15 × 15	20 × 15	20 × 20	30 × 15	30 × 20	50 × 15	50 × 20	100 × 20
RASCL	9.02	10.58	10.87	13.98	16.09	9.32	9.89	3.96

Table: Optimality gaps in % to the Taillard instances [33] of all strategies. For each of these 3 runs per strategy, the best optimality gap per problem size is printed in bold. The results of [6] are shown below for comparison.

Average makespan per PDR per problem size

The table contains the average makespans achieved by the corresponding PDR on 1,000 randomly generated instances of a problem size. The same set of instances were used for each PDR to ensure comparability. The lowest value is printed bold in each column.

	6x6	10x10	15x15	20x15	20x20	30x15	30x20	50x15	50x20	100x20
RND	7.7	14.7	23.7	28.1	33.1	36.6	42.1	52.3	59.1	97.0
SPT	11.5	29.6	65.8	85.9	117.5	126.8	174.6	209.7	287.6	571.3
LPT	11.6	30.0	65.9	86.1	118.3	126.8	174.8	209.9	287.5	571.4
MTR	**6.1**	**10.9**	**17.0**	**19.9**	**23.1**	**25.5**	**29.1**	**36.5**	**40.3**	**67.4**
LRPT	6.5	11.8	18.7	22.2	25.3	29.5	33.0	43.0	47.1	81.0

Table 6.1: Average makespan per PDR per problem size

Configurations

All constant parameters are shown in the following table.

Parameter	**Value**	**Description**
α_{actor}	10^{-4}	learning rate of actor model
α_{critic}	10^{-4}	learning rate of critic model
t_{top}	0.15	adaptive threshold
stepsize	4	*stepsize* for strategies: CCL, ICCL, SSCL, ISSCL
spacing	8	*spacing* for strategies: SSCL, ISSCL, HICL
u	100	rate of calculating the performance distribution g and level l
$rate_{GD}$	500	sampling rate for metric *gap difference*
$rate_{RG}$	150	sampling rate for metric *relative gap*

Table: Assigned values of the parameters

Name, Vorname: _____

E r k l ä r u n g

gem. § 15 Abs. 6 PO (Allgemeine Bestimmungen)

Hiermit erkläre ich, dass ich die von mir eingereichte Abschlussarbeit (Bachelor-Thesis) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Stellen der Abschlussarbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen wurden, in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Datum, Unterschrift: _____

E r k l ä r u n g

Hiermit erkläre ich micht damit einverstanden, dass meine Abschlussarbeit (Bachelor-Thesis) wissenschaftlich interessierten Personen oder Institutionen und im Rahmen von externen Qualitätssicherungsmaßnahmen des Studienganges zur Einsichtnahme zur Verfügung gestellt werden kann.
Korrektur- oder Bewertungshinsweise in meiner Arbeit dürfen nicht zitiert werden.

Datum, Unterschrift: _____