Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI-23)

On the Study of Curriculum Learning for Inferring Dispatching Policies on the Job Shop Scheduling

Zangir Iklassov , **Dmitrii Medvedev** , **Ruben Solozabal Ochoa de Retana** and **Martin Takac**
Mohamed bin Zayed University of Artificial Intelligence
zangir.iklassov@mbzuai.ac.ae, dmitrii.medvedev@mbzuai.ac.ae, ruben.solozabal@mbzuai.ac.ae,
martin.takac@mbzuai.ac.ae

Abstract

This paper studies the use of Curriculum Learning on Reinforcement Learning (RL) to improve the performance of the dispatching policies learned on the Job-shop Scheduling Problem (JSP). Current works in the literature present a large optimality gap when learning *end-to-end* solutions on this problem. In this regard, we identify the difficulty for RL to learn directly on large instances as part of the issue and use Curriculum Learning (CL) to mitigate this effect. Particularly, CL sequences the learning process in a *curriculum* of increasing complexity tasks, which allows learning on large instances that otherwise would be impossible to learn from scratch. In this paper, we present a size-agnostic model that enables us to demonstrate that current curriculum strategies have a major impact on the quality of the solution inferred. In addition, we introduce a novel Reinforced Adaptive Staircase Curriculum Learning (RASCL) strategy, which adjusts the difficulty level during the learning process by revisiting the worst-performing instances. Conducted experiments on Taillard's and Demirkol's datasets show that the presented approach significantly improves the current state-of-the-art models on the JSP. It reduces the average optimality gap from 19.35% to 10.46% on Taillard's instances and from 38.43% to 18.85% on Demirkol's instances.

1 Introduction

The Job-shop Scheduling Problem (JSP) is a combinatorial problem with vast implications on real-world tasks. It is formulated as a set of jobs, each consisting of a set of operations, to be processed on a set of heterogeneous machines. Furthermore, each operation has a specific machine assigned to it, and the operational time it takes to complete is known in advance. The goal is to define the scheduling order of the operations such that the total completion time or *makespan* is minimized. This problem applies to many tasks concerned with the optimal assignment of capital goods versus means of production, e.g., manufacturing, storage, transportation, etc. However, the JSP is an NP-hard problem, and it is therefore impractical to solve optimally. Optimization methods such as integer programming or constraint programming are computationally expensive methods that are unrealistic to use when fast solutions are required. Therefore, in practice, hand-engineered heuristics are commonly used to find approximate solutions to the problem.

However, designing such heuristic rules is a daunting task that requires specialized knowledge of the problem. In the case of the JSP, Priority Dispatch Rules (PDRs) are a family of constructive heuristics used on scheduling problems that are fast to compute and easy to implement. Although, it is not clear how to extrapolate these rules to different problems. Therefore, recent studies have focused on using Reinforcement Learning (RL) to derive domain-specific heuristics automatically. This technique is known in the literature as Neural Combinatorial Optimization (NCO). Unlike traditional PDRs that use the same dispatching rule for any instance in the problem, NCO particularizes a solution based on the instance information, which makes this technique very compelling. This technique has already demonstrated promising results in the JSP [Zhang *et al.*2020a, Wang *et al.*2021a].

This paper provides a nostrum for improving the learning strategy on the JSP using Curriculum Learning (CL). To this end, we design a problem-specific architecture and novel training approach that stresses this direction. Our main contributions are summarized below:

- This work particularizes a **deep learning model** to address the JSP. Particularly, we present an autoregressive model that iteratively constructs the solution based on the operations that are yet to be scheduled, as well as on the state of the machines during the resolution process. The model is equivariant w.r.t. the job information but also size-agnostic, which enables to seamlessly use it on different problem sizes and train it performing a curriculum strategy.
- We also present **Reinforced Adaptive Staircase Curriculum Learning (RASCL)**, a novel CL strategy that improves the model's performance based on the flexible assignment of difficulty levels during the learning process. In this work, we identify the difficulty for traditional RL methods to learn on large instances of the problem. To mitigate this issue, RASCL dynamically reinforces the model by revisiting the worst-performing instances. Conducted experiments show that this strategy

presents an improvement when compared to previous CL approaches.

The aforesaid ideas deliver an improvement in the scheduling solutions when compared to state-of-the-art works on JSP [Zhang *et al.*2020a]. Notably, we reduce the optimality gap from 19.35% to 10.46% on Taillard's instances and from 38.43% to 18.85% on Demirkol's instances[1].

The remainder of this paper is organized as follows. In Section 2, we present the state-of-the-art of NCO applied on the JSP. In Section 3, we provide a formal description of the method. Then, in Section 4 the architecture description, and in Section 5 the learning strategy are provided. In Section 6 the method is experimentally validated. Finally, Section 7 concludes the paper.

2 Related Work

The first attempts of using RL to address scheduling problems date back to the 90s [Mahadevan *et al.*1997, Mahadevan and Theocharous1998, Zhang and Dietterich1995]. Of specific relevance is Zhang and Dietterich's paper [Zhang and Dietterich1995] on allocating resources for NASA shuttle missions. As it was reflected in the literature, one of the advantages of using RL for such a purpose is that it can be seamlessly used for static, dynamic [Gabel and Riedmiller2008, Aydin and Öztemel2000], and stochastic variants of the problem. However, due to technological limitations, this technique could only be applied to small instances of the problem, limiting its applicability significantly.

The recent rise of deep learning has allowed extrapolating this technique to more realistic problem instances. Prior works have particularized deep neural networks, e.g., Pointer Networks [Vinyals *et al.*2015b], to learn in combinatorial spaces. In [Bello *et al.*2016], deep RL was implemented for the first time to learn *end-to-end* solutions to combinatorial problems. The authors used the Pointer Network in an actor-critic architecture to address the Travelling Salesman Problem. Further studies implemented Transformer networks [Deudon *et al.*2018, Vaswani *et al.*2017, Kool *et al.*2018]. However, all these works share in common that they are based on sequence-to-sequence models, where the complete solution is output at once. Thereby, heavy sampling and searching techniques were required at interference to improve the solution. Our approach is in line with [Nazari *et al.*2018], where the Vehicle Routing Problem is described as a Markov Decision Process, and the solution is iteratively constructed based on sequential decisions.

In the particular case of the JSP, several techniques have been applied to learn on the problem. Imitation learning was used in [Ingimundardottir and Runarsson2018] to learn from optimal solutions on training instances that were labeled using a Mixed-Integer Programming (MIP) solver. RL has also been applied to the problem, e.g., to select pre-defined candidate PDRs according to the scheduling conditions [Aydin and Öztemel2000, Lin *et al.*2019]. Other works have addressed the problem from a multi-agent perspective, e.g., in cooperative manufacturing [Gabel and Riedmiller2008, Waschneck *et al.*2018] where each agent controls a production line. Moreover, numerous examples of scheduling in many application domains, including manufacturing [Lin *et al.*2019, Wang *et al.*2021a], distributed computing [Mao *et al.*2019, Zhang *et al.*2020b, Sun *et al.*2021] or supply chains. Despite the effort, many of these approaches do not beat traditional heuristics. In addition, a major limitation in many of these works is that the state representation is hard-bounded by some factors (e.g., size of jobs or number of operations to consider), not enabling to scale the solution to arbitrary problem sizes.

Zhang et al. [Zhang *et al.*2020a] presented the first size-agnostic model on the problem. It formulates the JSP as a disjunctive graph and uses a high discriminative Graph Isomorphism Network (GIN) to embed the states in the resolution procedure. They prove to capture raw features from small problem instances and successfully generalize solutions to much larger instances that the ones experienced during the training. Even though graph neural networks have shown unprecedented success embedding not-euclidean spaces, in this paper, we show that a simpler autoregressive neural network is enough to better capture the state information and infer a dispatching rule.

One of the key features of the model presented in this work is the use of CL to stress the learning capabilities of the model to even larger problem instances. This aspect has been pointed out several times in the literature, e.g. [Zheng *et al.*2019] employs transfer learning to reconstruct the trained policies on problems of different sizes. However, policy transfer is still relatively costly and inconvenient. Also, [Liu *et al.*2020] uses lifelong learning, where an agent will not only learn to optimize one specific problem instance but reuse what it has learned from previous instances. They also proposed a parallel training method that combines asynchronous updates with a deep deterministic policy gradient to speed up model training. In [Lisicki *et al.*2020], the authors use a CL strategy with an adaptive staircase mechanism, where, at each iteration, the model changes the difficulty level according to the model performance. In our work, we build on top of this idea, tracking the model's behavior during the learning stressing the worst-performing levels.

3 Method

In the JSP, a finite set of m jobs $\{J_i\}_{i=1}^{m}$ are to be processed on a finite set of n machines $\{M_k\}_{k=1}^{n}$. Each job J_i consists of a sequence of n operations $O_{i,1} \rightarrow O_{i,2} \rightarrow ... \rightarrow O_{i,n}$, that have to be processed in a predetermined order. For each operation, $O_{i,j}$, the machine M_k is assigned for a given processing time $D_{i,j}$. The goal is to determine the scheduling order of the operations such that the total execution time or *makespan* is minimized.

There are several constraints that need to be taken into account when obtaining a solution on the problem:

- no-overlap constraints, determine that each machine shall process only one operation at a time;
- non-preemptive constraints, stating that once the process of an operation is initiated, it shall not be interrupted

[1]Taillard and DMU datasets are publicly available on http://optimizizer.com/TA.php and http://optimizizer.com/DMU.php respectively.

before completion;

- precedence constraints, establishing that 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.

3.1 Learning the Dispatching Policy

In this work, we formulate the resolution process on the JSP as a Markov Decision Process (MDP), in which the solution is iteratively constructed according to the dispatching decisions inferred. This process resembles how constructive heuristic algorithms operate. We are therefore concerned with learning the dispatching policy that based on the state on the resolution process infers the operation to schedule next. To this end, we propose learning an autoregressive model that dispatches the operations according to the scheduling policy π_θ, which is parameterized using a neural network and optimized using RL.

More formally, we consider that a solution to an instance $x \in \mathcal{X}$ of the JSP consists of $t \in \{1, ..., m \cdot n\}$ decision steps that correspond to the dispatching operations. We denoted as $s_t \in \mathcal{S}$ the state of the machinery, which evolves with the scheduling decisions at every step. The actions denoted as $a_t \in \mathcal{A}$ correspond to the job for which the next pending operation is dispatched at that step. The goal is therefore learning a dispatching policy $\pi : \mathcal{X} \times \mathcal{S} \rightarrow \Delta^{|\mathcal{A}|}$ that for an instance x at the resolution state s_t, outputs the probability distribution over the next pending operations. The transition function $P : \mathcal{X} \times \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ corresponds to the dynamics on the problem, which are deterministic. The model dynamics are therefore expressed as

$$s_{t+1} \sim P(\cdot|x, s_t, a_t) \qquad a_t \sim \pi(\cdot|x, s_t). \tag{1}$$

The process repeats until all operations are scheduled. At that point, the solution y is defined as the sequence of dispatch actions $(a_1, a_2, ..., a_{m \cdot n})$.

Regarding the cost function $C : \mathcal{X} \times \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, it is defined as the temporal increment in the completion time τ between two successive states

$$C_t(x, s_t, a_t) = \tau(x, s_{t+1}) - \tau(x, s_t), \tag{2}$$

where $\tau(x, s_t)$ refers to the time required to complete all operations from the initial state until s_t, which is computed using the Gantt diagram of a given instance. The cost function can be seen as the negative of the RL reward, and it is formulated to easily apply the *reward-to-go* trick [Sutton and Barto2018]. This uses causality to remove the past sum over rewards so that only rewards collected after the current decision step are considered, which helps in reducing the variance of policy gradient. The makespan $T(x)$ is therefore directly obtained as the sum of collected costs along the resolution process $T(x) = \sum_t C_t(x, s_t, a_t)$.

In order to learn the policy we resort to the Reinforce algorithm [Williams1992] and the model is optimized for minimizing the expected makespan,

$$\mathcal{J}^\pi(\theta) = \mathbb{E}_{x \sim \mathcal{X}, a \sim \pi_\theta(\cdot|x, s_t)} \left[T^{\pi_\theta}(x) \right]. \tag{3}$$

Following the common variation of *Policy Gradient Theorem* [Sutton *et al.*1999], the gradient of the objective function is obtained as

$$\nabla_\theta \hat{\mathcal{J}}^\pi(\theta) \approx \frac{1}{B} \sum_b \sum_{t=1}^{n \cdot m} \Big((G(x, s_t) - b_\phi(x, s_t)) \cdot \nabla_\theta \log \pi_\theta(a_t|x, s_t) \Big), \tag{4}$$

where $b_\phi(s_t, x)$ is the *baseline* - the estimate of the value function at state s_t of the problem instance x, and G, called *return*, is the actual cumulative cost starting at time step t and till all $n \cdot m$ operations are schedule $G(s_t, x) = \sum_t^{n \cdot m} C_t(s_t, a_t)$. The baseline is computed using a critic head and it is used to reduce the variance of the gradients, and therefore, to speed up the convergence. The critic is parameterized with parameters ϕ and it is trained to minimize

$$L(\phi) = \frac{1}{B} \sum_b \sum_{t=1}^{n \cdot m} ||b_\phi(x, s_t) - G(x, s_t)||^2 . \tag{5}$$

3.2 Inference Strategies

As reflected in the literature [Bello *et al.*2016, Kool *et al.*2018], the greedy inference of scheduling decisions from the policy function π_θ used to result in poor results in NCO. Therefore, inference strategies, although sometimes quite computationally expensive [Bello *et al.*2016], are commonly used to report competitive solutions. In the following, we describe the inference strategies considered in this paper:

Sampling. This technique produces several scheduling solutions by repeating the inference process directly sampling from the policy distribution. Sometimes a temperature hyperparameter is used to soften the output distribution and increase the diversity of the solutions. We denote as s the width of the sampling strategy.

POMO [Kwon *et al.*2020]. Policy Optimization with Multiple Optima mixes both, *greedy* and *sampling* inference strategies. At the initial state, this method rolls out p initial scheduling actions $\{a_0^{(1)}, ..., a_0^{(p)}\}$. Then, the solutions are completed using the greedy strategy. This creates p solutions to the problem $\{y^{(1)}, ..., y^{(p)}\}$ that are evaluated to select the best one. The authors demonstrate that this approach is competitive and computationally efficient compared to the *sampling* strategy.

Beam search [Joshi *et al.*2019, Wang *et al.*2021a]. This strategy explores a beam of b dispatches at each decision step. At the initial state, the b most likely scheduling actions $\{a_0^{(1)}, ..., a_0^{(b)}\}$ are selected. Then, at the next decision step for each candidate solution, a new set of b actions is rolled out, thus, the total number of available actions becomes $b \times b$. To prevent the number of candidate solutions from exploding in number, beam search only selects the b candidates with the highest likelihood at each step. At the end of the inference process, b candidate solutions are created.

4 Architecture Details

Top view of the architecture. The proposed architecture is depicted in Fig. 1. The model receives the instance description x as well as the internal state during the problem resolution s_t and outputs $\pi(x, s_t)$, the distribution over the next pending operations at the current decision step t. The state of the problem

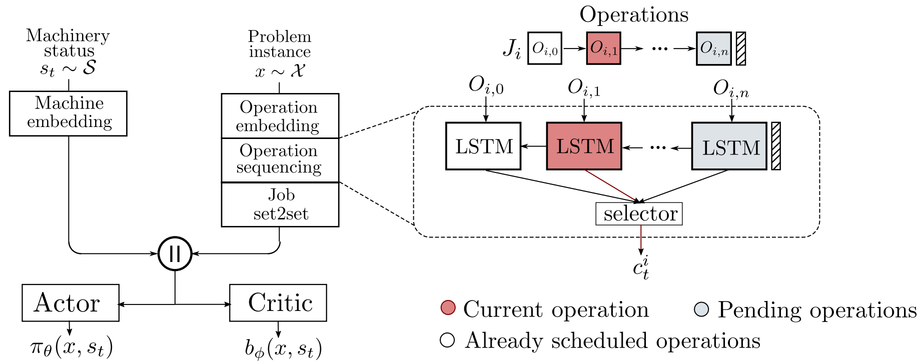


Figure 1: **Model Architecture.** The model receives the instance description x as well as the state of the machinery s_t at step t and outputs $\pi(x, s_t)$ a distribution over the jobs indicating the next pending operations to be scheduled. The operations on each job are recurrently encoded using an LSTM network, and a set2set network is used to combine the job information in an equivariant manner. The information regarding the machinery and job status is concatenated and connected to the actor and critic heads respectively.

consists of the following features: the index of the operations being currently processed, the status of machines and the remaining processing times. Regarding the job information, the sequencing of the operations for each job is obtained using a recurrent LSTM [Hochreiter and Schmidhuber1997] network. Once the operation sequencing is encoded, the job information is passed through a *set2set* module [Vinyals *et al.*2015a], which is particularly useful to aggregate the job descriptions in an equivariant manner. This module neglects the positioning of the jobs in the instance description, which increases the sample efficiency of the model as equivalent representations of the problem have the same representation. Finally, the embeddings of both descriptors are concatenated and passed into actor-critic networks to calculate the distribution over the dispatching actions, as well as the estimate of the return.

Operation sequencing. Considering that the dispatching actions at any given step only depend on the remaining operations, we use a LSTM network to capture this information. The LSTM operates backward, starting from the last operation in this job till the current operation. The embeding for the operation sequence is computed once for the instance description, and its result is stored for being used during the resolution process. Particularly, the index of the operation currently processed is used to select the output of the LSTM corresponding to the current step t. We denote this vector c_t^i and it encodes the information of the pending operations for the job J_i.

Decoding the solution. The solution is sequentially constructed in an autoregressive process where at each step the model indicates the operation to dispatch next. This is done by sampling from the probability distribution the policy-head computes. The model tracks therefore the operations already scheduled and only operates over the next available operation in each job. Once the scheduling order is determined, we can compute the makespan. The model needs to track also when all the operations in a job are scheduled, as selecting the job is no longer a valid action. To this end, we use a masking scheme that operates on the policy-head and reduces to zero the chances of selecting an invalid dispatch.

Algorithm 1 RASCL algorithm

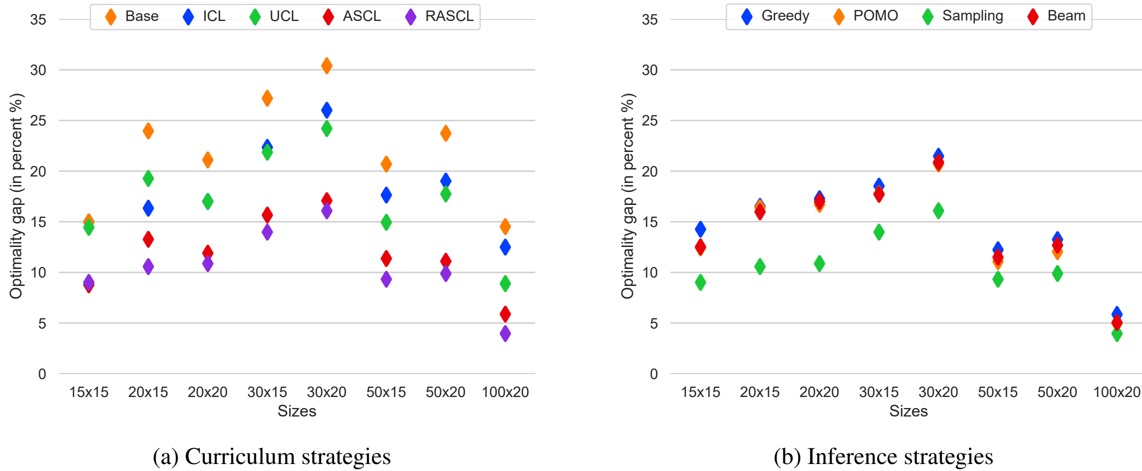
Data: Define the levels of difficulty $\mathcal{L} = \{l_0,...,l_K\}$ and the optimality threshold δ_{th}
Initialize $L = \{l_0\}$, $l = l_0$, and $k = 0$
while $k \leq K$ **do**
  Update $\{g(l) \mid l \in L\}$ according to the optimality gaps on the testing dataset.
  if $\max\{g(l)\} \leq \delta_{th}$ **then**
    Increase the difficulty $k = k + 1$ and $L = L \cup \{l_k\}$
  else
    Sample level $l \sim$ softmax$(\{1 / g(l) \mid l \in L\})$
  end
  Train policy π_θ on level l
end
Result: Trained policy π_θ

5 Learning Using a Curriculum Strategy

Curriculum Learning (CL) is a methodology that uses scenarios of increasing complexity to train the model. CL was adopted from social and educational systems to build a curricula of increasing complexity [Wang *et al.*2021b]. Such an approach allows the model to learn policies from the easiest scenario that can be extrapolated to harder levels of difficulty. In the case of the JSP, we identify the difficulty levels as the different sizes on the problem $m \times n$. The bigger the number of jobs m and operations n, the harder is to learn. In this work, we consider the following most popular CL strategies:

Incremental curriculum learning (ICL). In CL a set of levels of difficulty $\mathcal{L} = \{l_0,...,l_K\}$ has to be predefined for performing the training. Incremental CL represents the simplest approach and it sequentially trains the model on increasing difficulty levels in a sequential manner. The drawback of this strategy is the model's tendency to catastrophic forgetting when trained on consecutive grades of complexity [Lisicki *et al.*2020], as previous levels of difficulty are not revisited.

Uniform curriculum learning (UCL). In uniform CL, at each iteration the model selects the training complexity with equal chances from the different difficulty levels. In this way, the model is exposed to various training levels and the policy stays inherent to different problem sizes. However, based on



(a) Curriculum strategies

(b) Inference strategies

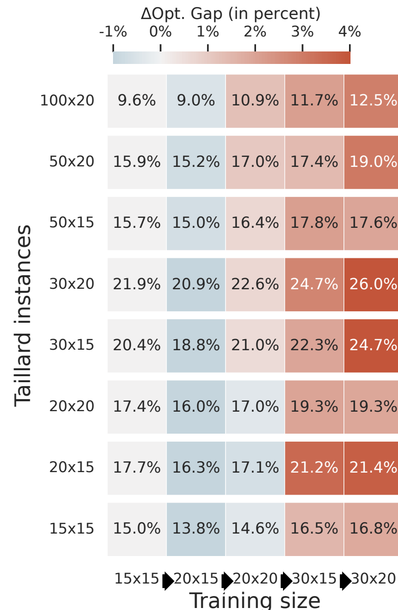
Figure 2: Optimatlity gap (in percent) on the (a) curriculum strategies and (b) inference strategies on Taillard's instances. In (a), all strategies incorporate a sampling (s=128). In (b), the measurements are obtained on the RASCL model for POMO (p=3), sampling (s=128) and Beam search (b=3).

the work of [Lisicki *et al.*2020], learning on small-size tasks is more accessible rather than sampling each training level from a uniform distribution of all sizes of interest, which eventually may not be the most helpful learning strategy.

Adaptive staircase curriculum learning (ASCL). Adaptive staircase is introduced to RL in [Lisicki *et al.*2020]. This strategy introduces flexibility in selecting the level of complexity during the training. Specifically, the model initializes the learning process on the easiest difficulty level l_0 and adaptively increases the difficulty based on the observed optimality gap. At the level l_k, ASCL revisits previous difficulty levels to not 'forget' its corresponding policies. The levels to revisit are selected are randomly selected following a uniform distribution.

Reinforced adaptive staircase CL (RASCL). In this work we propose a methodology for dynamically adjusting the difficulty between the levels. Similar to ASCL, the agent's is initiated at the lowest difficulty level l_0 with the aim to make its way to the top level l_K. However, the limitation of ASCL is inherent in the randomness of revisiting the previous levels.

To tackle the problem, RASCL proposes a further improvement of ASCL by revisiting the levels where model shows the worst performance. Specifically, RASCL determines the level of difficulty comparing the estimation of the return to the optimal one. To this end, this method evaluates the optimality gap $g(l)$ for each difficulty level and establishes a policy for revisiting levels inversely proportional to the gap observed. The bigger the optimality gap achieved in a level the higher the chances to revisit the level. The complete algorithm description can be found in Algorithm 1. Particularly, the method tracks a subset of levels L from where the training is performed. The subset of levels available starts as $L = \{l_0\}$ and it increases during the learning. Once the set of levels L are mastered, L is augmented incorporating the next difficulty level. The condition for incorporating the next level is by comparing the maximum optimality gap on L to a predetermined threshold δ_{th}. If the model's return is below the threshold, RASCL adds the new level of difficulty. Otherwise, the method keeps training sampling instances from the current subset of levels L. The proposed algorithm reinforces the learned policy by revisiting and anchoring the most problematic difficulty levels.



Taillard instances	15x15	20x15	20x20	30x15	30x20
100x20	9.6%	9.0%	10.9%	11.7%	12.5%
50x20	15.9%	15.2%	17.0%	17.4%	19.0%
50x15	15.7%	15.0%	16.4%	17.8%	17.6%
30x20	21.9%	20.9%	22.6%	24.7%	26.0%
30x15	20.4%	18.8%	21.0%	22.3%	24.7%
20x20	17.4%	16.0%	17.0%	19.3%	19.3%
20x15	17.7%	16.3%	17.1%	21.2%	21.4%
15x15	15.0%	13.8%	14.6%	16.5%	16.8%

Figure 3: Increment in the optimality gap (in percent) on Taillard's instances when trained using (a) Base and (b) ICL learning strategies. From (a), learning directly on large instances is hard for the model; whereas in (b), progressively incrementing the problem size during training reduces the optimality gap.

6 Experimentation

Datasets. We train and evaluate our model on scheduling instances of the sizes used on Taillard's [Taillard1993] and

Instances		SPT	FDD/WKR	MWKR	MOPNR	[Zhang *et al.*2020a]	RASCL (s=128)
15×15	Obj.	1546.1	1808.6	1464.3	1481.3	1547.4	**1339.8**
	Gap	(25.89%)	(47.15%)	(19.15%)	(20.53%)	(25.96%)	**(9.02%)**
20×15	Obj.	1813.5	2054.0	1683.6	1686.7	1774.7	**1509.3**
	Gap	(32.82%)	(50.57%)	(23.35%)	(23.55%)	(30.03%)	**(10.58%)**
20×20	Obj.	2067.0	2387.2	1969.8	1968.3	2128.1	**1793.1**
	Gap	(27.75%)	(47.61%)	(21.81%)	(21.71%)	(31.61%)	**(10.87%)**
30×15	Obj.	2419.3	2590.8	2214.8	2195.8	2378.8	**2038.1**
	Gap	(35.27%)	(45.02%)	(23.91%)	(22.83%)	(33.0%)	**(13.98%)**
30×20	Obj.	2619.1	3045.0	2439.0	2433.6	2603.9	**2261.5**
	Gap	(34.44%)	(56.3%)	(25.17%)	(24.94%)	(33.62%)	**(16.09%)**
50×15	Obj.	3441.0	3736.3	3240.0	3254.5	3393.8	**3030.8**
	Gap	(24.11%)	(34.77%)	(16.86%)	(17.37%)	(22.38%)	**(9.32%)**
50×20	Obj.	3570.8	4022.1	3352.8	3346.9	3593.9	**3125.1**
	Gap	(25.54%)	(41.5%)	(17.95%)	(17.68%)	(26.51%)	**(9.89%)**
100×20	Obj.	6139.0	6620.7	5812.2	5856.9	6097.6	**5578.9**
	Gap	(14.41%)	(23.39%)	(8.31%)	(9.15%)	(13.61%)	**(3.96%)**

(a) Taillard's instances

Instances		SPT	FDD/WKR	MWKR	MOPNR	[Zhang *et al.*2020a]	RASCL (s=128)
20×15	Obj.	4951.5	4666.3	4909.9	4513.2	4215.3	**3610.0**
	Gap	(64.13%)	(53.57%)	(62.15%)	(49.16%)	(38.95%)	**(19.36%)**
20×20	Obj.	5690.5	5298.2	5489.0	5052.3	4804.5	**4028.9**
	Gap	(64.57%)	(52.52%)	(58.16%)	(45.17%)	(37.74%)	**(15.98%)**
30×15	Obj.	6306.2	6016.5	6252.9	5742.8	5557.9	**4522.0**
	Gap	(62.57%)	(54.12%)	(60.95%)	(47.14%)	(41.86%)	**(16.35%)**
30×20	Obj.	7036.0	6827.3	6925.0	6491.9	5967.4	**5106.0**
	Gap	(65.91%)	(60.09%)	(63.16%)	(51.97%)	(39.48%)	**(20.0%)**
40×15	Obj.	7601.2	7420.0	7484.2	7105.5	6663.9	**5731.9**
	Gap	(55.88%)	(51.42%)	(52.87%)	(44.72%)	(35.38%)	**(17.49%)**
40×20	Obj.	8538.1	8210.9	8460.9	7870.7	7375.8	**6584.1**
	Gap	(63.0%)	(55.52%)	(61.11%)	(49.22%)	(39.38%)	**(25.42%)**
50×15	Obj.	8975.4	9150.2	8906.0	8436.5	8179.4	**7242.1**
	Gap	(50.37%)	(52.53%)	(48.93%)	(40.79%)	(36.2%)	**(21.54%)**
50×20	Obj.	10132.8	9899.6	9807.0	9408.0	8751.6	**7176.9**
	Gap	(62.2%)	(57.26%)	(56.4%)	(49.61%)	(38.86%)	**(14.66%)**

(b) DMU's instances

Table 1: Results on (a) Taillard's and (b) DMU instances. **Objective** indicates the average *makespan* for a given problem size; and **Gap**, the average difference (in percent) to the upper bound known for the instances.

	Instances	Greedy	**POMO** [Kwon *et al.*2020] (p=3)	**Sampling** (s=128)	**Beam** (b=3)
TAILLARD	15×15	1404.2 (14.26%)	1381.9 (12.43%)	**1339.8 (9.02%)**	1382.9 (12.52%)
	20×15	1590.3 (16.52%)	1588.6 (16.40%)	**1509.3 (10.58%)**	1582.9 (15.97%)
	20×20	1896.5 (17.27%)	1887.5 (16.71%)	**1793.1 (10.87%)**	1892.1 (17.01%)
	30×15	2118.9 (18.52%)	2105.0 (17.78%)	**2038.1 (13.98%)**	2104.8 (17.7%)
	30×20	2365.7 (21.47%)	2350.8 (20.70%)	**2261.5 (16.09%)**	2354.0 (20.87%)
	50×15	3111.6 (12.23%)	3079.1 (11.08%)	**3030.8 (9.32%)**	3090.9(11.5%)
	50×20	3219.6 (13.24%)	3185.9 (12.05%)	**3125.1 (9.89%)**	3203.4 (12.68%)
	100×20	5680.9 (5.86%)	5633.3 (4.97%)	**5578.9 (3.96%)**	5637.5 (5.05%)

Table 2: Inference strategies for RASCL on Taillard's instances. Results are reported as average **Objective (Gap)**.

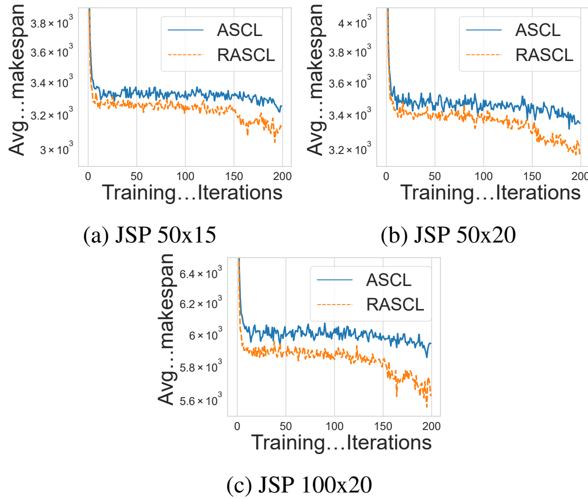
Demirkol's (DMU) [Demirkol *et al.*1998] datasets: 15x15, 20x15, 20x20, 30x20 and 30x15. The instances are expressed as JSP $m \times n$, where m represents the number of jobs, and n is the number of operations. To test the model's performance, we use substantially larger instances that are not included in the training set: 50x15, 50x20, and 100x20.

Baselines. For the choice of *baselines*, we resort to the analysis on priority dispatch rules (PDRs) in [Sels *et al.*2012] and choose the most popular ones in the research community: Shortest Processing Time (SPT), Minimum Ratio of Flow Due Date to Most Work Remaining (FDD/WKR), Most Work Remaining (MWKR), Most Operations Remaining (MOPNR). Among neural combinatorial solvers, we use the latest state-of-the-art results of the graph neural model on the public JSP benchmarks presented in [Zhang *et al.*2020a]. For approximating the optimal solution, required when estimating the optimality gap in RASCL, we resort to the constraint programming solver of Google OR-Tools CP-SAT. In the following, we refer to the *Base model* as architecture presented in Fig. 1 trained for a particular size of the JSP, without performing a curriculum.

First, we train five instances of the base model on five sizes: 15x15, 20x15, 20x20, 30x15, and 30x20. Each instance of the base model is dedicated to one of five problem sizes. Then, we test all base model instances on an extended set of sizes, adding 50x15, 50x20, and 100x20. The results reflect that the model achieves best performance when trained on the smallest size 15x15, where it shows the smallest optimality gap on all testing sizes, from 14.98% for 15x15 to 9.58% for 100x20. This reflects the difficulty for the model to directly learn on larger instances of the problem.

The model's results can be improved further, as the base model trained on a small size can hardly capture all combinatorial space of action policies inherent in large-size problems. To validate this hypothesis we incorporate different CL methods in the training. We start with ICL, for which we train five instances of the base model in the following manner: the model assigned to the 15x15 size is trained on this size only, the model assigned to 20x15 is trained sequentially on the previous size, which in this case 15x15, and on its assigned 20x15 size. The same logic applies for all sizes up to 30x20, and each learning cycle takes n iterations. Fig. 3 shows the results of this approach. Here, the 20x15 model, being trained consecutively on 15x15 and 20x15 sizes, exhibits the best performance reducing the average gap across all sizes tested. However, the ICL approach has an obvious drawback. The best result is still achieved on a small-size problem, 20x15, meaning that learning on larger-size instances is still hard.

To address the problem, we test three additional CL strategies including our proposed RASCL, see Fig. 2(a). One can see that UCL generally shows near-ICL behavior, while ASCL and RASCL models demonstrate notably better performance. Particularly, RASCL shows the smallest optimality gap on all test sizes, except for 15x15, where ASCL slightly outper-



(a) JSP 50x15

(b) JSP 50x20

(c) JSP 100x20

Figure 4: Average makespan observed during the training for (a) 50x15 (b) 50x20, and (c) 100x20 sizes. RASCL finds more competitive solutions when compared to ASCL, and stresses the learning process during more iterations.

forms RASCL by 0.24%. The reinforced approach displays the smallest optimality gap from 10.58% for 20x15 instances to 3.96% for 100x20 instances.

We also provide in Table 2 a comparison for different inference strategies: Greedy, POMO, Sampling and Beam search, all applied to the RASCL model. POMO $(p = 3)$ and Beam $(s = 3)$, show very similar results, only slightly better than greedy inference. The sampling strategy $(s = 128)$ shows the best result across all test sizes, and, consequently, it is the inference strategy used in the following experiments.

At this point, we summarize the results using RASCL at learning and Sampling as the inference strategy. Table 1(a) provides the comparison results on Taillard's dataset. RASCL model displays robust behavior on all test sizes, improving previous best results [Zhang *et al.*2020a] on average by 45.94%. This model shows a maximum optimality gap of 16.09% on Taillard's dataset, which is a much tighter bound than the offered by traditional PDRs.

To further test our model, we refer to DMU dataset consisting of 80 instances from 20x15 to 50x20 sizes. Table 1(b) shows outperforming behavior of RASCL model again. On average, it improves previous best results by 50.95%.

7 Conclusions and Future Work

In this work, we present a deep-RL model to automatically learn a dispatching policy on the JSP. To this end, we formulate the resolution process as MDP, in which a solution is iteratively constructed based on intermediate states on the resolution process. In this work, we present a particularized deep learning model on the JSP that is size-agnostic and also equivariant w.r.t. the job information, which benefits the training as equivalent instances are similarly encoded. In order to improve the behavior of the model, this work relies on Curriculum Learning. In this direction, we present a novel RASCL strategy, which dynamically adjusts the difficulty of the learning according to the model's performance during the learning process. Experiments on Taillard's and Demirkol's instances show that our model improves the optimality gap w.r.t. the current state-of-the-art model by 45.94% and 50.95%, respectively. Our results corroborate that learning using a curricula is key for improving the results on the problem.

There are several future directions for this work. For example, we would like to remark the potential of this technology for addressing stochastic or partially observable problems, domains where traditional approaches have shown limited capabilities.

Hyper-parameter	Value
Instance input dimension	$2 \times m \times n$
Machinery state input dimension	$2 \times n$
Encoder dimension	128
LSTM [layer, dim]	[1, 128]
Actor/Critic [depth,width]	[2, 16]
Activation function	ReLU
Learning rate	1E-04
Optimizer	Adam
Batch size	128
Training iterations (per level) - n	45000
Level evaluation (iterations)	100_{th}
Threshold opt. gap - δ_{th}	15%

Table 3: Hyper-parameter values.

A Models and Configurations

We train the models using the same set of hyperparameters for every size on the problem (see Table 3). The validation set consists of 1,000 randomly generated JSP instances. Both the actor and critic utilized fully-connected networks with two layers: the first layer contained 16 neurons, while the second layer contained m neurons for the Actor and a single neuron for the Critic. ReLU activation was used for all layers. The training was conducted using the Adam optimizer with a learning rate of 10^{-4} and a batch size of 128. The Base, ICL and UCL models are trained for 45,000 iterations. Each model including ASCL and RASCL is tested every 100_{th} iteration. During training, for ASCL and RASCL, a maximum gap of δ_{th} = 15% between the optimal test solutions and the model's solutions was used as the criterion for proceeding to the next level. Figure 4 shows the average makespan results for 50x15, 50x20, and 100x20 size instances during the training. Regarding the inference strategy, a tree-width of 3 is used for the Beam search and POMO selection strategies, and a sample size of 128 is used for the sampling strategy. Our implementation is available online.[2] Detailed instructions for reproducing the training, evaluation, and plotting the results are included. The model is implemented in PyTorch, and the JSP environment was created using the Gym library. The training was conducted on an NVIDIA A100 SXM 40GB GPU with 2x AMD EPYC 7742 CPUs (8 cores) and 256GB RAM. To train the RASCL model from scratch, it required approximately 8 hours using the described hardware.

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

References

[Aydin and Öztemel, 2000] M Emin Aydin and Ercan Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2-3):169–178, 2000.

[Bello *et al.*, 2016] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[Demirkol *et al.*, 1998] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, August 1998.

[Deudon *et al.*, 2018] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. In *International conference on the integration of constraint programming, artificial intelligence, and operations research*, pages 170–181. Springer, 2018.

[Gabel and Riedmiller, 2008] Thomas Gabel and Martin Riedmiller. Adaptive reactive job-shop scheduling with reinforcement learning agents. *International Journal of Information Technology and Intelligent Computing*, 24(4):14–18, 2008.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[Ingimundardottir and Runarsson, 2018] Helga Ingimundardottir and Thomas Philip Runarsson. Discovering dispatching rules from data using imitation learning: A case study for the job-shop problem. *Journal of Scheduling*, 21(4):413–428, 2018.

[Joshi *et al.*, 2019] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

[Kool *et al.*, 2018] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[Kwon *et al.*, 2020] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

[Lin *et al.*, 2019] Chun-Cheng Lin, Der-Jiunn Deng, Yen-Ling Chih, and Hsin-Ting Chiu. Smart manufacturing scheduling with edge computing using multiclass deep q network. *IEEE Transactions on Industrial Informatics*, 15(7):4276–4284, 2019.

[Lisicki *et al.*, 2020] Michal Lisicki, Arash Afkanpour, and Graham W Taylor. Evaluating curriculum learning strategies in neural combinatorial optimization. *arXiv preprint arXiv:2011.06188*, 2020.

[Liu *et al.*, 2020] Chien-Liang Liu, Chuan-Chin Chang, and Chun-Jan Tseng. Actor-critic deep reinforcement learning for solving job shop scheduling problems. *Ieee Access*, 8:71752–71762, 2020.

[Mahadevan and Theocharous, 1998] Sridhar Mahadevan and Georgios Theocharous. Optimizing production manufacturing using reinforcement learning. In *FLAIRS conference*, volume 372, page 377, 1998.

[Mahadevan *et al.*, 1997] Sridhar Mahadevan, Nicholas Marchalleck, Tapas K Das, and Abhijit Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Machine Learning Interantional Workshop*, pages 202–210, 1997.

[Mao *et al.*, 2019] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288. 2019.

[Nazari *et al.*, 2018] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V Snyder, and Martin Takáč. Reinforcement learning for solving the vehicle routing problem. In *Conference on Neural Information Processing Systems, NeurIPS 2018*, 2018.

[Sels *et al.*, 2012] Veronique Sels, Nele Gheysen, and Mario Vanhoucke. A comparison of priority rules for the job shop scheduling problem under different flow time-and tardiness-related objective functions. *International Journal of Production Research*, 50(15):4255–4270, 2012.

[Sun *et al.*, 2021] Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 3314–3320, 2021.

[Sutton and Barto, 2018] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sutton *et al.*, 1999] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[Taillard, 1993] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, January 1993.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[Vinyals *et al.*, 2015a] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[Vinyals *et al.*, 2015b] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[Wang *et al.*, 2021a] Libing Wang, Xin Hu, Yin Wang, Sujie Xu, Shijun Ma, Kexin Yang, Zhijun Liu, and Weidong Wang. Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning. *Computer Networks*, 190:107969, 2021.

[Wang *et al.*, 2021b] Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[Waschneck *et al.*, 2018] Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, and Andreas Kyek. Optimization of global production scheduling with deep reinforcement learning. *Procedia Cirp*, 72:1264–1269, 2018.

[Williams, 1992] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[Zhang and Dietterich, 1995] Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120. Citeseer, 1995.

[Zhang *et al.*, 2020a] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1621–1632, 2020.

[Zhang *et al.*, 2020b] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: an automated hpc batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[Zheng *et al.*, 2019] Shuai Zheng, Chetan Gupta, and Susumu Serita. Manufacturing dispatching using reinforcement and transfer learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 655–671. Springer, 2019.