2nd Conference on Production Systems and Logistics

Discovering Heuristics And Metaheuristics For Job Shop Scheduling From Scratch Via Deep Reinforcement Learning

Tilo van Ekeris[1], Richard Meyes[1], Tobias Meisen[1]

[1]*Chair of Technologies and Management of Digital Transformation, University of Wuppertal, Germany*

Abstract

Scheduling is the mathematical problem of allocating tasks to resources considering certain constraints. The goal is to achieve the best possible scheduling quality given a quality metric like *makespan*. Typical scheduling problems, including the classic *Job Shop Scheduling Problem* (JSP or JSSP), are *NP-hard*; meaning it is infeasible to use optimal solvers for big problem sizes. Instead, heuristics are frequently used to find suboptimal solutions in polynomial time, especially in real-world applications. Recently, *Deep Reinforcement Learning* (DRL) has also been applied to find solutions for planning problems like the JSP. In DRL, agents learn solution strategies for specific problem classes through the principle of trial and error. In this paper, we explore the connection between known heuristics and DRL: Heuristics always rely on features that can be extracted from the considered problem with low computational effort. We show that DRL agents, for which we limit the available observation to the underlying features of well-known heuristics, learn the behaviour of the more qualitative heuristics from scratch, while they do not learn the behaviour of less qualitative heuristics that would also be possible learning outcomes given the same feature as observation. Additionally, we motivate the use of DRL as a metaheuristic generator by training with the features of multiple basic heuristics. We show promising results that indicate that this learned metaheuristic finds better schedules in terms of *makespan* than any single simple heuristic – while only requiring simple computations in the time-critical solution phase and thus being faster than optimal solvers.

Keywords

Deep Reinforcement Learning (DRL); Production Planning; Scheduling; Job Shop Scheduling (JSP, JSSP); Proximal Policy Optimization (PPO); Heuristics; Metaheuristics

1. Introduction

In production and logistics, scheduling is the mathematical problem of allocating tasks to resources considering certain constraints. Different types of scheduling problems occur, the formulations range from static formulations like the *Job Shop Scheduling Problem* (JSP, JSSP) to formulations that include dynamics (e.g., newly appearing jobs [1] or machines with changing availability) or other complexities (e.g., the requirement to schedule tooling in addition to machines). Typical scheduling problems are combinatorial optimization problems and finding their solutions is proven to be *NP-hard* [2]. Classically, scheduling problems are solved by either optimal solvers, basic heuristics or metaheuristics. In this work we seek to apply *Deep Reinforcement Learning* (DRL) to scheduling problems. We focus on the following JSP formulation: The goal of the problem is to produce a schedule with minimal total production duration *(makespan)* by assigning a number of jobs n_j, divided into multiple tasks with fixed durations, to a fixed number of machines n_m (denoted a $n_j \times n_m$ JSP). The machine required for each task is given by the problem. The constraints for scheduling are: C1) the tasks within each job have to be scheduled in the given order; C2) only one task can be scheduled on each machine in each time-step. Figure 1a shows an example JSP with four jobs and four machines. An optimal solution regarding this JSP with *makespan* 24 is depicted in Figure 1b. Other optimal solutions can be trivially created by moving tasks which are not lying on the critical path (e.g., in Figure 1b, *Task 2-3* can be moved by up to three timesteps to the right without changing the resulting *makespan*).

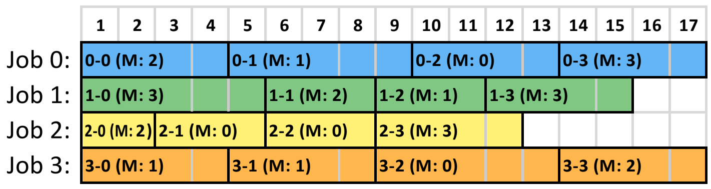

Figure 1a: Example of a 4 × 4 JSP: Each of the four jobs contains four tasks with a unique identifier ("id(j)-id(t)") with id(j) the job number and id(t) the task number within its job as well as the required machine ("M: id(m)") and the duration which is represented by the length in blocks

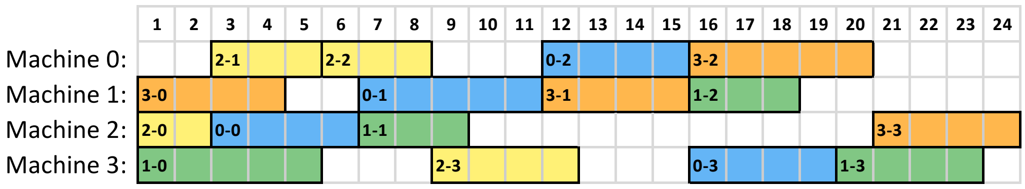

Figure 1b: (Example) optimal solution for the JSP from Figure 1a calculated with the Google OR tools solver

The objective of our study was twofold: First, we examined whether a DRL approach is able to discover basic heuristics for solving the JSP from scratch given a limited feature observation (Experiment E1). Second, we trained a DRL agent on a set of features to let the agent construct a scheduling strategy on these features. We then tested this strategy against basic heuristics and an optimal solver from a quality as well as from a run time perspective (Experiment E2).

2. Background and Related Work

2.1 Classic solution methods

For optimal solving, the JSP inputs and constraints can be expressed mathematically through a set of (in)equations and then put into state-of-the art solvers like the *CP-SAT solver* from the *Google OR Tools* suite [3], where CP indicates *Constraint Programming* and *SAT* the *Boolean satisfiability problem*. The *CP-SAT solver* uses a technique called *Lazy Clause Generation* [4] to find optimal solutions even to *NP-hard* problems as efficiently as possible.

Heuristics are frequently used, especially in real-world applications, where the available time for scheduling is often limited. Heuristics rely on features that are calculated from the problem with low computational effort so that heuristics' run times are typically low – but the simplifying nature of heuristics does not guarantee optimal or even good schedules in terms of a given quality metric. Research in heuristics is a vast field with a long history. Pinedo [5] discusses various approaches, from simple heuristics like *Earliest Due Date* (EDD), and *Earliest Release Date* (ERD) to other techniques such as *genetic algorithms* and *ant colony optimization*.

Finally, metaheuristics "combine basic heuristic methods in higher level frameworks aimed at efficiently and effectively exploring a search space" [6]. They try to get the best of two worlds: Use diverse features to try to use better planning decisions while maintaining a low computational effort.

2.2 Deep Reinforcement Learning

DRL is a machine learning discipline combining *Deep Learning* (DL) with *Reinforcement Learning* (RL) in which artificial agents are trained to take good actions given an observation based on the state of the environment [7]. Agents are trained by exposing them to a large amount of observations and giving them a reward signal after individual actions and/or a chain of actions (sparse rewards). The agents in DRL consist of at least one neural network that maps the observations to actions (cf. Figure 2).

DRL has recently seen the biggest breakthroughs in Games like classical board games Shogi, Chess and Go [8] as well as real-time strategy games like StarCraft II [9] because computer games and simulations allow to generate the large amounts of training data required by DRL more easily than real-world settings. Within the field of DRL, different training algorithms have been developed. The algorithm used in this work, *Proximal Policy Optimization* (PPO) [10], is widely used, for example by OpenAI in their work on the game DotA2, OpenAI Five [11]. PPO is a policy gradient algorithm which uses two separate neural networks, a value network and a policy network, as opposed to the algorithm *Deep Q-Network* (DQN), which uses only a value network [12].

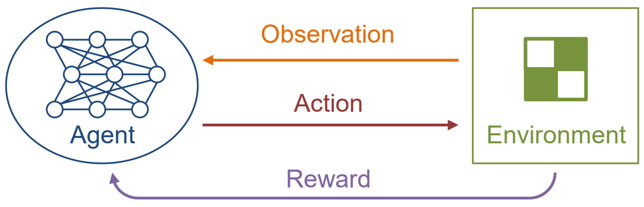

Figure 2: Deep Reinforcement Learning approach

More recently, DRL has been applied to a diverse range of problems outside of games, including scheduling problems. Liu et al. [13] trained a DQN on the JSP and they formulated the agent's action as a choice from a set of fixed heuristics. This does not allow the agent to find its own heuristic from the underlying features. Luo [14] took a similar approach: DQN is used (with the enhancements *double DQN* and *soft target weight update*) and the agent's action consists of choosing from "six composite rules [which] are designed to simultaneously determine which operation to process next". Waschneck et al. [15] also apply DQN to production scheduling and construct a more complex multi-agent setup to solve their custom scheduling problem. The scheduling quality is not better than heuristics, but comparable to a human expert benchmark. Rinciog et al. [16] apply the AlphaGo Zero algorithm [17] to a JSP with multiple processing steps and compare the results to the simple heuristic *Earliest Due Date* (EDD) as well as to the classic search algorithm *Monte Carlo Tree Search* (MCTS). The results are better than EDD after Fine Tuning the RL agent by a small margin and better than MCTS by a larger margin (taking into account that MCTS was only run with a small number of roll-outs in the vast search tree of the scheduling problem). Finally, Kuhnle et al. [18] give a good overview about relevant research approaches and other related work. They also solve a dispatching problem with several groups of machines using the *Trust Region Policy Optimization* (TRPO) algorithm [19]. They extensively discuss the problem between sparse and dense rewards and describe the modelling of their RL agent in a detailed manner.

Although several groups of researchers have conducted experiments training DRL agents on scheduling problems, to our best knowledge there are no studies that explicitly examine what DRL agents learn given only basic feature observations with the connected heuristics.

3. Methods

This section describes the design of the DRL agent, the data generation, agent training and agent testing in the order that is typically followed when applying DRL to a specific problem.

3.1 Design of features and the agent's observation

In order to be able to learn, a DRL agent needs an observation that is calculated from the state of the environment (cf. Figure 2). In our experiments, this observation consists of a number (depending on the experiment) of feature vectors that are calculated from the environment state and each have one entry for each job, so the vectors have length n_j. In the following, we introduce the features used. Some are inspired by known basic heuristics and some are our own developments:

The *remaining job duration* RJD_j is defined by the sum of the durations of all unscheduled tasks from a job j (cf. Figure 3). Two well-known basic heuristics defined on this feature are: The *Longest Remaining Duration* (LRD) and the *Shortest Remaining Duration* (SRD), which at each iteration choose to schedule the next task from the job with the longest respectively shortest RJD. For example, the SRD heuristic would choose *Job 2* in Figure 3. Please note that a large number of heuristic rules can be defined on this feature (not only max/min).

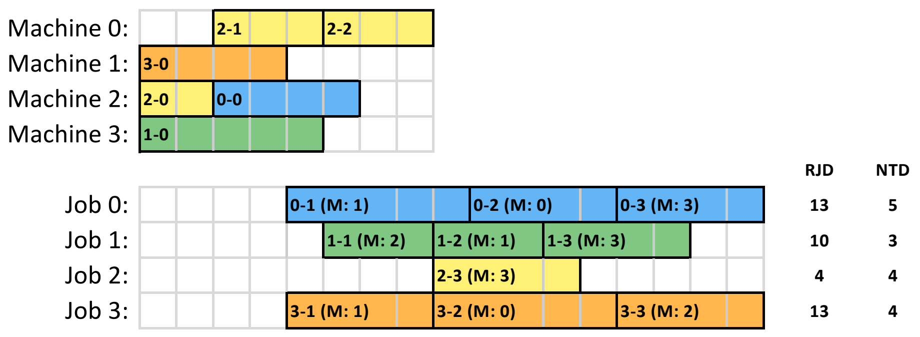

Figure 3: Partially scheduled example JSP from Figure 1a (top) with calculated features "remaining job duration (RJD)" and "next task's duration (NTD)" for the tasks remaining to be scheduled (bottom)

The *next task's duration* NTD_j is defined by the duration of the next task of a job j (cf. Figure 3). Two well-known basic heuristics defined on this feature are: The *Longest Processing Time* (LPT) and the *Shortest Processing Time* (SPT), which at each iteration choose to schedule the next task from the job with the longest respectively shortest NTD. For example, the SPT heuristic would choose *Job 1* in Figure 3.

The *remaining task count* RTC_j counts the number of unscheduled tasks of a job j.

The *bottleneck feature* BF_j seeks to guide the agent to choose tasks from jobs that mitigate machine bottlenecks. First, to determine a proxy of how likely a machine is to become a bottleneck, it sums up the duration of unscheduled tasks *per machine* (as opposed to *per job*). This vector is multiplied with a (n_m, n_j)-matrix where each item equals the sum of the unscheduled duration within job j to be scheduled on machine m.

3.2 Design of the agent's actions

In DRL, it needs to be defined how the agent is able to interact with the environment via actions (cf. Figure 2). We decided that the agent chooses from which job to schedule the next task. It cannot choose specific tasks; always the first unscheduled task from the chosen job is scheduled. Thus, the action is a number $a \in \mathbb{N}\,; 0 \leq a < n_j$. This action is then transferred to a scheduling component that schedules the resulting task at the earliest possible timestep complying with the two JSP constraints (cf. Section 1).

3.3 Design of the agent's reward

We seek to minimize the *achieved makespan* m_a of the completed schedule and the reward needs to be designed so that the agent is steered towards this goal. Because we can evaluate m_a only after the JSP is completely scheduled, we deal with so-called *sparse rewards.* The lowest possible or *optimal makespan* m_o of the problems is unknown (without running an optimal solver like the CP-SAT solver), so we compare m_a to a *lower bound makespan* m_{lb} which is calculated by simply adding all the task durations per machine and taking the maximum. Please note that $m_{lb} \leq m_o$ is satisfied for all possible JSPs, because $m_{lb} = m_o$ if and only if an optimal solution has no empty timesteps on the machine with the longest total duration and $m_{lb} < m_o$ otherwise. When m_a approaches m_{lb}, the reward shall rise, and with a greater slope for diminishing differences. We therefore decided to calculate the reward via a negative logarithm with the ratio between m_a and m_{lb} as argument. In order to leave room for punishing the agent for unwanted behaviour, we added constant shifts and scalings. The final definition of reward r is given in Formula (1) and plotted in Figure 4.

$$r = \left(-\ln\left(\frac{m_a}{m_{lb}} - 0.95\right) + 2.5\right) * 50 \tag{1}$$

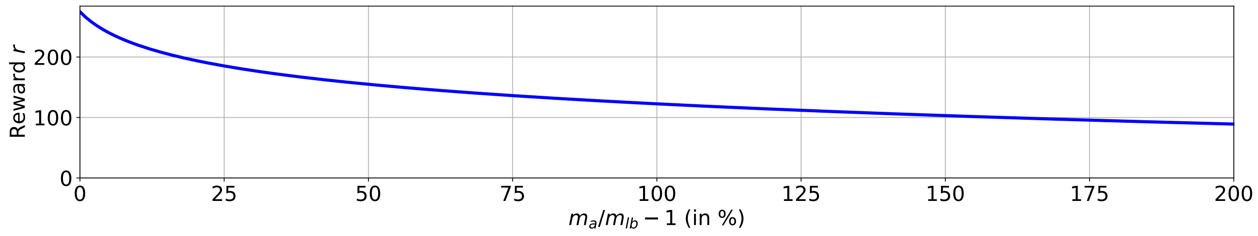

Figure 4: Reward r for successful solutions of the JSP as a function of m_a and m_{lb}

If the agent chooses a job that is already completely scheduled, it receives a large negative reward $r = -200.0$. The goal is that the agent learns that it should never choose such a job. This works in most of the cases, however, depending on the number of training timesteps, it is possible that the agent does not fully learn this behaviour which can result in an “action loop”, where the agent always takes the “wrong” job while accumulating negative rewards. This behaviour was also described by Kuhnle et al. [18]. It can be mitigated by either detecting the action loop and forcing the agent to take a valid decision or choose an appropriate number of training timesteps for which this behaviour does not appear.

3.4 Problem classes and data generation

Depending on how exactly the JSPs are constructed, different problem classes are generated. As we know from extensive research in the ML community [20], the data that is used to train models and agents can (and usually will) introduce biases in the resulting behavior. We generate JSPs in a reproducible way, but we are aware that different choices could be taken that generate different JSP problem classes.

First, we initialize a random number generator with a seed. For a JSP with n_j jobs and n_m machines, we took the choice that the JSP is constructed so that each job also contains n_j tasks, so that the problems grow in two “dimensions” when augmenting n_j. For each task, the required machine is determined by sampling from a uniform distribution. For the duration d, we tested two different generation methods. In the first, we generate d by sampling from a normal distribution $d \sim \mathcal{N}(n_j, (0.25 \cdot n_j)^2)$ with mean n_j and standard deviation $0.25 \cdot n_j$ (discretizing and forcing $d \geq 1$). In the second, we sample d from a discrete uniform distribution $d \sim \mathcal{U}(1, n_j)$. This generates already two different problem classes of JSPs that we have both tested in our experiments.

3.5 Agent training

The training is done by exposing the agent to the generated training JSPs. In each timestep, the observation that is required for the experiment is calculated, normalized and passed to the agent. The agent chooses its action and might receive a reward directly (in the case of the negative reward due to choosing the wrong job), then receives the next observation. After the schedule for the current problem is finished, the agent receives the reward as described in Section 3.3 and is subsequently presented the next JSP. Our DLR agents are all trained with the PPO [10] algorithm. This algorithm trains two deep neural networks, a *policy network* and a *value network* (*Actor-Critic architecture*). For our problem sizes we kept the default network architecture of the Stable Baselines implementation [21], which defines the policy network and the value network as completely separate (no shared neurons) with two layers consisting of 64 neurons each. We kept the hyperparameters *learning_rate=0.0003*, *gamma=0.99* and *clip_range=0.2* at the proposed defaults. The agents were trained for 250,000 timesteps (one timestep being one *Reinforcement Learning* cycle of observation, action and reward) on JSP training data and tested on 1,000 JSPs from a different test set. Each JSP from the training set was only used once.

3.6 Baseline schedulers and agent testing

In order to test the DRL agent against baseline methods, we have implemented several other schedulers. To get the optimal solution, we use Google's OR tools JSP solver [3]. We also test against well-known standard heuristics (cf. Section 3.1) as well as other simple heuristics, like a random solver. Testing is done by letting all schedulers solve the JSPs from the test set and comparing the resulting schedules. To compare the scheduler similarity, we compare the produced schedules one-by-one and calculate their distances by taking the sum of absolute differences of the task starting times (in discrete timesteps) for each task, divided by the total number of tasks. We call this distance measure *cumulative absolute task deviation* (CATD). To examine the absolute and relative scheduling quality, we measure the *achieved makespan* m_a for every schedule and for each scheduler calculate the mean, minimum, maximum and standard deviation of the distribution of *makespans* m_a and run times (excluding set-up times for all algorithms).

4. Experiments and Results

4.1 Discovering basic heuristics from basic features from scratch (Experiment E1)

In this experiment we examine if the DRL agent is able to discover known basic heuristics from scratch when given only the heuristics' feature as observation and which of the possible heuristics it learns. We have run the experiment (independently) for the two features RJD and NTD (cf. Section 3.1). Figures 5 and 6 show the distances of schedules (cf. Section 3.6) for a selection of scheduler combinations.

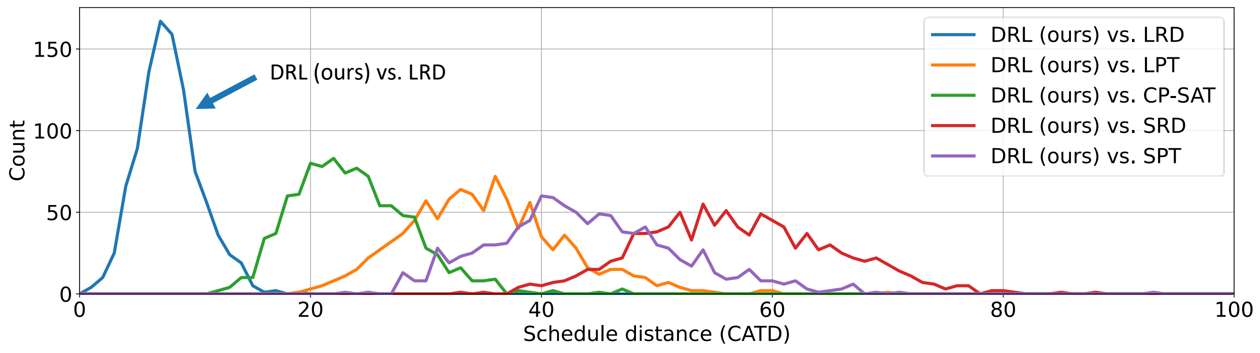

Figure 5: Schedule distances (CATD) histogram comparing our DRL scheduler (trained with the RJD feature as the only observation) against baseline schedulers

Each graph is a histogram that shows the distances between the schedules of two schedulers. A graph that is located more towards the origin means that more schedules generated by the two schedulers for the same problem are similar, while a graph located more towards larger arguments means that the schedules are less similar. The results indicate that the DRL agent is able to discover known heuristics from scratch given the underlying basic feature. For the feature RJD, the agent learns the heuristic LRD and not SRD, which both rely on the same feature, presumably because LRD is the better performing heuristic in terms of *makespan* (cf. Section 4.2). For the feature NTD, the similarity of the DRL scheduler and the LPT scheduler is greater than for any other heuristic (including SPT), but the results are less obvious than in the RJD case. Our explanation for this is that for the feature NTD, neither the heuristic LPT nor the heuristic SPT produce really good schedules (in terms of *makespan*, cf. Section 4.2) and the DRL agent thus is not rewarded to either clearly learn the behaviour of LPT nor SPT.

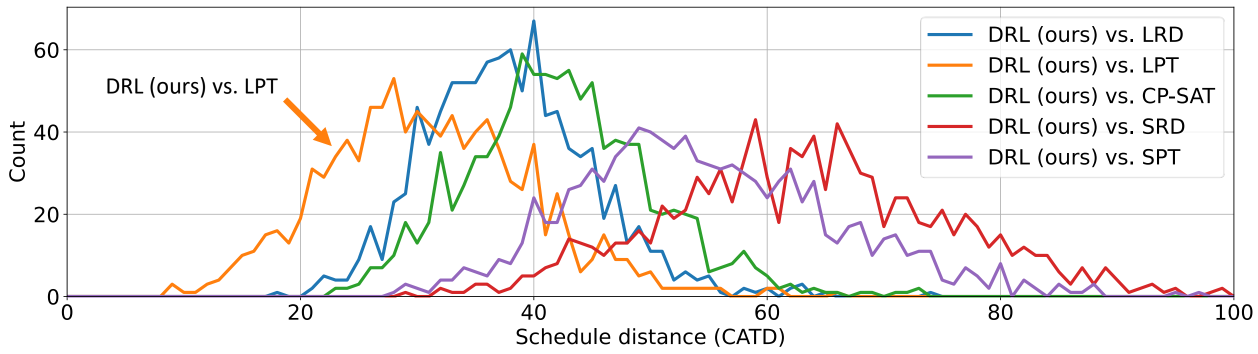

Figure 6: Schedule distances (CATD) histogram comparing our DRL scheduler (trained with the NTD feature as the only observation) against baseline schedulers

4.2 Learning a new metaheuristic from multiple basic features from scratch (Experiment E2)

In this experiment we trained DRL agents on JSP problems of sizes 8x8 and 12x12 using a set of features as the agent's observation (as opposed to experiment E1, where the agent's observation only consists of a single feature in each experiment). An exhaustive study has been performed in which we have found that an observation consisting of the three features RJD, RTC and BF leads to the best results (in terms of *makespan*). See Tables 1 and 2 for the results on 8x8 and 12x12 JSPs respectively, both with normal task duration distribution (cf. Section 3.4).

Table 1: *Achieved makespans* m_a and solver run times on 1000 8x8 JSPs

	Achieved makespan m_a				Run time per JSP (in ms)			
	mean	min	max	stdev	mean	min	max	stdev
CP-SAT (optimal)	107.9	86	164	10.9	64.3	27.6	412.3	28.8
DRL (ours)	**120.9**	**91**	**169**	**10.9**	**298.5**	**257.3**	**823.0**	**32.6**
LRD	121.3	96	168	10.9	11.0	9.7	22.3	1.0
Random	137.3	108	230	13.2	2.2	1.9	5.8	0.2
LPT	142.6	107	198	14.3	3.0	2.6	5.2	0.3
SPT	148.9	109	231	16.5	8.6	7.7	15.7	0.7
SRD	158.2	115	226	17.8	17.0	15.1	183.8	6.7

The results indicate that learning a good metaheuristic from scratch from a set of basic features is possible. From a quality perspective, we achieve *makespans*, that are on average lower (better) than those from the best single heuristic. From a run time perspective, we achieve lower run times than the optimal solver for 12x12 JSPs. When growing the problem size from 8x8 to 12x12, we see more than 20x increase in the mean run time of the optimal solver, but less than 3x increase for the DRL solver. We argue that this difference is due to the *NP-hardness* of the problem and that the effect would amplify with growing problem sizes. We thus find a compromise position of solution quality and run time that lies between heuristics and optimal solver.

Table 2: *Achieved makespans* m_a and solver run times on 1000 12x12 JSPs

	Achieved makespan m_a				Run time per JSP (in ms)			
	mean	min	max	stdev	mean	min	max	stdev
CP-SAT (optimal)	236.8	200	371	17.5	1518.3	123.2	33825.7	2608.4
DRL (ours)	**275.4**	**233**	**376**	**18.1**	**763.6**	**712.1**	**1981.2**	**58.8**
LRD	277.2	234	391	18.5	39.0	36.2	76.9	2.9
Random	316.0	258	419	23.9	5.8	5.1	10.4	0.4
LPT	337.3	261	455	28.6	7.6	6.7	11.8	0.5
SPT	350.3	276	519	30.1	17.5	16.0	27.9	1.2
SRD	374.0	294	540	32.6	48.9	45.4	82.2	3.3

Please note, that the run times between the simple heuristics LRD and SRD (analogously LPT and SPT) result from slightly more complex code to obtain a minimum value while ignoring values that equal zero in comparison with obtaining a maximum. We found that just putting more of the simple features to the DRL agent does not help to find schedules with a better quality (the contrary is true). Intuition suggests, that the DRL agent is capable to completely learn which features to weigh more or less so that more features should not decrease the scheduling quality. But at least this is not the case if the number of training timesteps and/or the network architecture/size remain unchanged. We did not observe significant differences between the two tested problem classes (cf. Section 3.4), even if an agent was trained on data with normal duration distribution and tested on data with uniform duration distribution.

5. Discussion and Outlook

In this paper we have shown two distinct results: First, a DRL agent is able to learn basic heuristics from scratch given underlying basic features. The results indicate that the agent behaves similarly to the best performing heuristic that can be defined on the feature. Second, a DRL agent can discover a good metaheuristic automatically, given multiple basic features. It generates lower *makespans* than any single simple heuristic. While this solution method is not competitive compared to heuristics from a run time perspective, it requires much less computational effort than an optimal solver. Of course, we would like to improve the quality margin of the DRL agent compared to the best simple heuristic. This is important to be able to completely justify the use of DRL with its higher runtime and additional training effort for scheduling problems. Thus, one path to follow is to increase the number of basic features that are presented to the DRL agent, trying to find the features that lead to better results. We would like to examine the JSPs that can currently only be solved with large errors in comparison to the optimal solver in order to learn something about the structure and to produce new feature vectors that might help the agent. Another idea is to use generative processes (e.g., *Generative Adversarial Networks*, GANs) to produce more of those “hard” JSP instances so that we can augment the amount of those JSPs in the training data set with the goal that the DRL agent can learn more from those instances. We would also like to further evaluate the impact of other reward function definitions.

The decision logic of the metaheuristic discovered by the DRL agent is hidden inside a neural network and could be made visible via “explainable AI” techniques in future work.

Currently, we must apply feature engineering to select the best set of features as the agent's observation. We would like to approach *end-2-end DRL* for scheduling problems, meaning to find a representation that describes the structure of the problem directly and can be fed into the DRL agent.

In our approach, we can possibly generalize to the number of machines, but not to the number of jobs, because the agent requires the same observation vector size as well as a fixed action vector size (both depending on the number of jobs) determined during training. In the future we would like to look into applying *Graph Neural Networks* (GNNs, like in [22] or [23] for the TSP) to solve JSPs, because the GNN architecture offers the possibility to generalize to larger problem instances.

Our far-stretched goal is to apply DRL to real-world production scheduling scenarios. Therefore, we would also like to approach dynamic scheduling problems (like in [1]) in the future, in which new jobs arrive or jobs can get deleted dynamically and machines can have defects so that they cannot be used in certain timesteps. Currently, we only consider a reward solely based on *makespan*. But in real-world scenarios, other goals come into play. Specifically, in the future we would like to include plan robustness into the reward. Our agent should not only be able to minimize *makespan*, but also produce schedules that are subjected to minimal change in the case that jobs, machines or other resources become unavailable. Finally, we would also like to tackle more complex scheduling problems that exist in the real world, e.g., including tooling changes within production that need to be scheduled along with the job scheduling.

Acknowledgements

This research work was undertaken within the research project AlphaMES [24] funded by the German Federal Ministry for Economics and Energy (BMWi).

Appendix

The Python framework *jobshop* that includes all our design choices in code and can be used to produce all the described results and diagrams will be published under an open source license upon publication.

References

[1] Fang, J., Xi, Y., 1997, A rolling horizon job shop rescheduling strategy in the dynamic environment, The International Journal of Advanced Manufacturing Technology, 13/3:227–232, DOI:10.1007/BF01305874.

[2] Gonzalez, T., 1982, Unit Execution Time Shop Problems, Mathematics of Operations Research, 7/1:57–66, DOI:10.1287/moor.7.1.57.

[3] Perron, L., Furnon, V., 2019, OR-Tools Ver. 7.2. Google, [Online]. https://developers.google.com/optimization/.

[4] Stuckey, P. J., 2010, Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving, in Integration of AI and OR Techniques in CP for CO Problems, Berlin, Heidelberg, pp. 5–9.

[5] Pinedo, M. L., 2016, Scheduling: Theory, Algorithms, and Systems, 5th ed. Springer International Publishing.

[6] Blum, C., Roli, A., 2001, Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, ACM Comput. Surv., 35:268–308, DOI:10.1145/937503.937505.

[7] Sutton, R. S., Barto, A. G., 2018, Reinforcement Learning: An Introduction. Cambridge, MA, USA.

[8] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., et al., 2018, A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, Science, 362/6419:1140–1144, DOI:10.1126/science.aar6404.

[9] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., et al., 2019, Grandmaster level in StarCraft II using multi-agent reinforcement learning, Nature, 575/7782:350–354, DOI:10.1038/s41586-019-1724-z.

[10] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017, Proximal Policy Optimization Algorithms, arXiv:1707.06347 [cs].

[11] OpenAI, Berner, C., Brockman, G., Chan, B., Cheung, V., et al., 2019, Dota 2 with Large Scale Deep Reinforcement Learning, arXiv:1912.06680 [cs, stat].

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., et al., 2015, Human-level control through deep reinforcement learning, Nature, DOI:10.1038/nature14236.

[13] Liu, C.-L., Chang, C.-C., Tseng, C.-J., 2020, Actor-Critic Deep Reinforcement Learning for Solving Job Shop Scheduling Problems, IEEE Access, 8:71752–71762, DOI:10.1109/ACCESS.2020.2987820.

[14] Luo, S., 2020, Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning, Applied Soft Computing, 91:106208, DOI:10.1016/j.asoc.2020.106208.

[15] Waschneck, B., Reichstaller, A., Belzner, L., Altenmüller, T., Bauernhansl, T., et al., 2018, Optimization of global production scheduling with deep reinforcement learning, Procedia CIRP, 72:1264–1269, DOI:10.1016/j.procir.2018.03.212.

[16] Rinciog, A., Mieth, C., Scheikl, P. M., Meyer, A., 2020, Sheet-Metal Production Scheduling Using AlphaGo Zero, DOI:10.15488/9676.

[17] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., et al., 2017, Mastering the game of Go without human knowledge, Nature, 550/7676:354–359, DOI:10.1038/nature24270.

[18] Kuhnle, A., Kaiser, J.-P., Theiß, F., Stricker, N., Lanza, G., 2020, Designing an adaptive production control system using reinforcement learning, Journal of Intelligent Manufacturing, DOI:10.1007/s10845-020-01612-y.

[19] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., Abbeel, P., 2017, Trust Region Policy Optimization, arXiv:1502.05477 [cs].

[20] Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., Galstyan, A., 2019, A Survey on Bias and Fairness in Machine Learning, arXiv:1908.09635 [cs].

[21] Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., et al., 2019, Stable Baselines3. https://github.com/DLR-RM/stable-baselines3.

[22] Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P. S., et al., Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning, p. 12.

[23] Joshi, C. K., Cappart, Q., Rousseau, L.-M., Laurent, T., Bresson, X., 2020, Learning TSP Requires Rethinking Generalization, arXiv:2006.07054 [cs, stat].

[24] van Ekeris, T., AlphaMES - TMDT – Bergische Universität Wuppertal. [Online]. https://www.tmdt.uni-wuppertal.de/de/projekte/alphames.html.

Biography

Tilo van Ekeris (*1986) has graduated with a Diploma degree from RWTH Aachen in Computer Engineering and a M. Sc. degree from Ecole Centrale Paris. He worked for 8 years for a top management consultancy company and several IT companies. Tilo worked at the TMDT during 2020/21 where he was involved in a range of Machine Learning/Artificial Intelligence projects.

Richard Meyes (*1989) has graduated as M.Sc. in Physics from RWTH Aachen and has worked as a research scientist in neuroscience and medicine as well as artificial intelligence and deep learning applications for industrial use cases for more than seven years. He is currently research group leader at the TMDT for explainable AI for industrial sensor data analytics.

Tobias Meisen (*1981) is Professor of Digital Transformation Technologies and Management at the University of Wuppertal since September 2018. He is also the Institute Director of the In-Institute for Systems Research in Information, Communication and Media Technology, the vice-chair of the Interdisciplinary Centre for Data Analytics and Machine Learning and a co-founder of Hotsprings GmbH.